X/Open CAE Specification

Window Management (X11R5):

Xlib - C Language Binding

X/Open Company Ltd.

Contents

Contents

Contents

List of Figures

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

* *CAE Specifications*

    CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

    Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

    CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```
This will return the index of publications for which Corrigenda exist.

This Document

This document is a CAE Specification (see above).

This X/Open interface definition is based upon the **X Window System Protocol** specification. It includes compatibility functions for earlier releases. It is also is based upon information contained in the following document:

> Xlib - C Language X Interface
> MIT X Consortium Standard, X Version 11, Release 5
> by Jim Gettys, MIT Project Athena, Ron Newman, MIT Project Athena,
> and Robert Scheifler, MIT Laboratory for Computer Science

Although this standard defines the programmatic interface, Xlib, to the X Window System core protocol, it does not attempt to define the format and semantics of the protocol itself. The definitive exposition of the protocol is contained in the **X Window System Protocol** specification and the documents it refers to.

Only the Xlib - C language binding has been considered in this document. To aid portability of applications that wish to use the Xlib interface, a default set of font names, color database entries and key symbol table entries is also defined.

Enhancements have been made to X11 Release 5, but the X11R5 specifications are upwards-compatible from the X11R4 specifications.

Structure

The source document for this publication, the **Xlib** - **C Language X Interface** specification from MIT, has undergone revision since X11R4, and this is reflected in the new or rearranged chapters of this publication.

- **Chapter 1** gives an overview of the X Window System, and is common to all 4 of the X/Open Window Management specifications.

- **Chapter 2**, **Introduction to Xlib**, includes part of the MIT standard Chapter 1 and some new X/Open material.

- **Chapters 3 to 17** of this specification are the same as Chapters 2 to 16 of the MIT standard. Compared with the CAE Specification for X11R4, readers will find that Chapters 6, 7 and 8

and Chapters 13, 14 and 16 are new or re-arranged.

- **Chapter 18**, **Additional X/Open Requirements**, describes additional X/Open requirements, over and above the X Consortium specification.

- are the same as these appendices in the MIT standard.

X/Open Window Management Document Set

This specification is one of four specifications in the X/Open Window Management (X11R5) document set. The full set comprises:

- X Window System Protocol

- Xlib - C Language Binding

- X Toolkit Intrinsics

- File Formats and Application Conventions.

These X11R5 specifications are available as a 4-volume set (Document Number T410).

The following table shows the structure and organisation of material in this document set in terms of the MIT documentation of the X Window System, on which the X/Open document set is based.

In each document, Chapter 1 is an X/Open overview of the X Window System, which is not in the MIT documentation.

X/Open Document	Subject	MIT Document
X Window System Protocol	Description and definition of the X Protocol	X Window System Protocol
Xlib - C Language Binding Chapters 2-17 and Appendices A-D	Description of Xlib functions and their use	Xlib - C Language X Interface
Chapter 18	X/Open additional requirements	None
X Toolkit Intrinsics	Description of X Toolkit functions and their use	X Toolkit Intrinsics
File Formats and Application Conventions	Various formats and conventions for application cooperation and communication	Inter-Client Communication Conventions Manual (ICCCM), Version 1.1 X Logical Font Description (XLFD), Version 1.4 Compound Text, Version 1.1 Bitmap Distribution Format (BDF) 2.1

The X Window Management (X11 Release 5) System is required by the X/Open Common Desktop Environment (XCDE), which defines a common graphical user interface environment. The other specifications in the XCDE family are:

- X/Open Common Desktop Environment (XCDE) 2-volume set comprising:

  — Definitions and Infrastructure

  — Services and Applications

- Motif Toolkit API (electronic publication)
- Calendaring and Scheduling API (XCS).

Trade Marks

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open® is a registered trade mark, and the ''X'' device is a trade mark, of X/Open Company Limited.

X Window System^{TM} is a trade mark of the Massachusetts Institute of Technology.

Acknowledgements

Referenced Documents

The following documents are referenced in this specification:

X11R5 X Protocol

X/Open CAE Specification, May 1995, Window Management (X11R5): X Window System Protocol (ISBN: 1-85912-087-3, C507).

X11R5 X Toolkit

X/Open CAE Specification, May 1995, Window Management (X11R5): X Toolkit Intrinsics (ISBN: 1-85912-089-X, C509).

X11R5 File Formats

X/Open CAE Specification, May 1995, Window Management (X11R5): File Formats and Applications Conventions (ISBN: 1-85912-090-3, C510).

This comprises:

— Inter-Client Communications Conventions Manual (ICCCM)

— X Logical Font Description (XLFD)

— Compound Text

— Bitmap Distribution Format (BDF).

ISO 8859-1

ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

Overview of the X Window System

1.1 Introduction

The X Window System is a network-transparent windowing system developed under the auspices of Project Athena at the Massachusetts Institute of Technology. The X Window System is implemented as a client-server model.' The window system functionality is provided by a display server, which is resident on a machine which has one or more monochrome or color raster displays attached. Client applications which require window system services attach to a server, and subsequently communicate with it, via an Inter-Process Communications connection. This uses a standard and extensible asynchronous protocol to communicate window system protocol requests to the server.

A client may, but not necessarily, run on the same machine as the X Server it is connected to. Applications may reside on hosts remotely connected to the system which hosts the display server by some kind of local or wide-area networking technology. This is dependent upon the level of functionality provided by the particular networking environment in which particular server and client implementations operate.

An X Window System server supports one or more physical, monochrome or color, raster streens, which display a logical hierarchy of (possibly) overlapping rectangular areas known as ''windows''. Also associated with the server is a number of input devices. Normally these include a keyboard and some form of pointing device, such as a mouse or digitising tablet.

At the top, or root, of the logical window hierarchy, is the ''root window'' which completely covers the physical screen with which the hierarchy is associated. In the normal course of operation, each ''root window'' will be partially, or completely, covered by ''child windows'' created by clients. Due to the organisation of the window hierarchy, an application program may create a tree of arbitrary depth on each screen. The X Window System Protocol provides applications with the functionality to create and manipulate windows and their associated attributes. The X Window System also provides the ability to associate arbitrary data with a window, access fonts and colors, perform general graphical output, and obtain input from the available devices, using a canonical, programmatic interface, which embodies a high degree of device independence.

A client that converses with the server using the X Window System protocol may operate ''correctly'' in isolation, but might not coexist properly with other clients sharing the same server. The **ICCCM** specification is a set of conventions to allow clients to cooperate in the areas of selections, cut buffers, window management, session management and resources.

1.2   X Window System Overview

The X Window System architecture is divided into two distinct parts (Figure 1-1):

display servers   Provide display capabilities and keep track of user input.

clients   Application programs that perform specific tasks.

This separation allows the clients and servers either to work together on the same system, or across a network. Regardless of where the clients are running, all user input and displayed output will occur on the workstation server. Communication is accomplished (in a network transparent fashion) using the X Protocol.
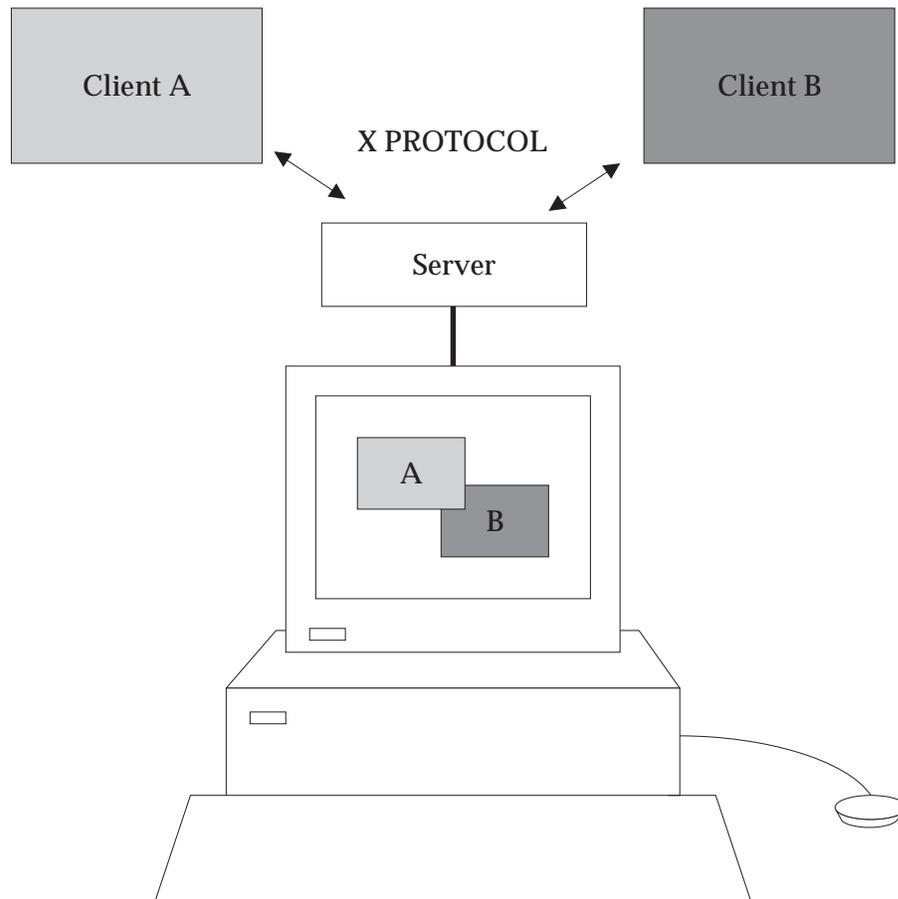
Figure 1-**1** X Window System Overview

1.2.1   X Platform Abstraction Layers

The X Window System consists of several distinct parts. Figure 1-2 shows them as layers.

- The *X Protocol* defines the format and sequencing of byte streams and semantics (messages) passed between X Clients and the X Server.

- *Xlib* specifies the function call interface to build the messages defined by the X Protocol.

- The *Xt Intrinsics* provide the basic constructs to support the creation and use of user interface objects (widgets).

- The *Widgets* provide a set of user interface features (such as menus and pushbuttons) and allow applications to manipulate these features using object-oriented programming techniques.

Widgets
Xt Intrinsics
Xlib
X Protocol
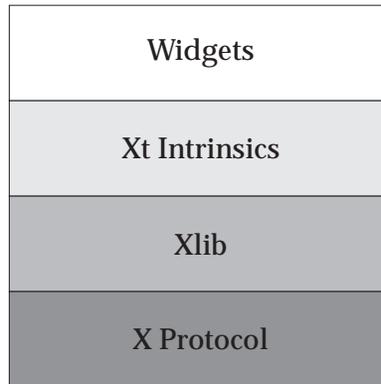
Figure 1-**2** X Platform Abstraction Layers

1.2.2   User Interface Platform

From the programmer's perspective, the X Window System provides a *User Interface Platform* with multiple interfaces (Figure 1-3). Applications can be developed using any or all of these interfaces, depending on the requirements of the developer. It is important to note here that the lowest-level interface is Xlib – the X Protocol does not provide a practical programming interface. Therefore, all interaction with the X Protocol is handled by Xlib calls. It is not necessary to program directly using Xlib to create an X Window System application. Therefore, the interface boundaries should be viewed as transparent from a programmer's perspective (the programmer may use any or all of them to achieve the desired results in the program).

```
                         ↓  ↓ ↓
      ┌──────────────────────────────┐
      │                              │
      │          Widgets             │
      │                              │
      │............↓.................│
      │                              │     Layers are
      │        Xt Intrinsics         │     Transparent
      │..............↓...............│
      │                              │
      │          Xlib                │
      │                              │
      └──────────────↑───────────────┘
                     │  X Protocol
                     │
                     └──────────→
```
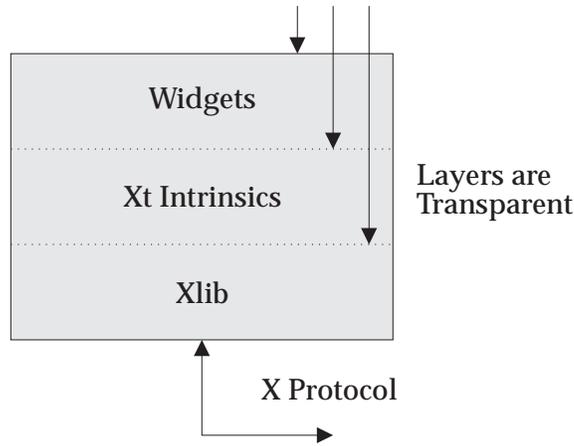
Figure 1-**3** User Interface Platform

1.2.3   A Single X Application

The User Interface platform provides all the services necessary to manage the user interface aspects of the application. *Application functionality* is that part of the application which is independent of any user interface function, but it is the application that knows what it wants to accomplish through the user interface. The translation of the application's user interface needs into user interface actions or displays is achieved through a form of binding.

This binding can be an integral part of the application, indistinguishable from the application functionality, or it can be a separate module created by a development tool or language and stored in a separate library or binary module. The separation of *application functionality* from *user interface functionality* (in so far as it is possible) helps to provide application portability and ease of maintenance.
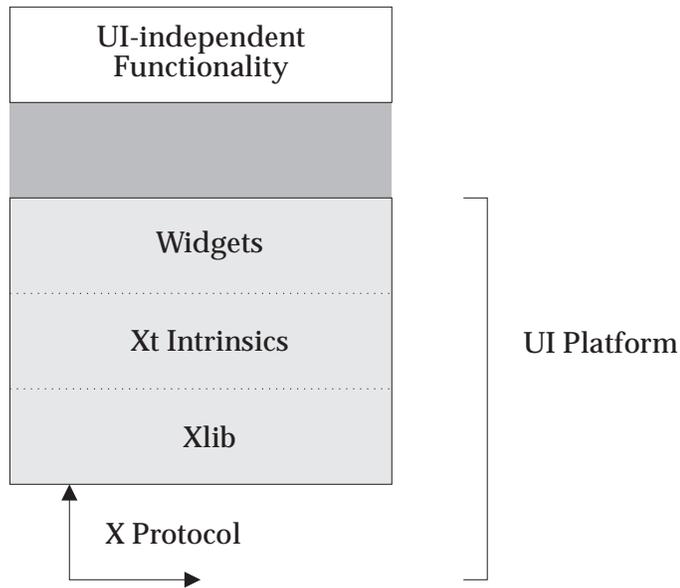
Figure 1-**4** A Single X Application

1.2.4 X Application Relationships

The X Window System environment usually consists of several *client applications*, all communicating with an *X Server* at the same time using the *X Protocol* (Figure 1-5). Some of these clients have special roles within the environment, such as window and session managers. In order for all of these applications to work together *cooperatively*, *Inter-Client Communications Conventions* have been established. These ensure that client applications will cooperate in their use of the server and can also interact directly with each other.

APPLICATION

APPLICATION

APPLICATION

X SERVER

WINDOW
MANAGER

APPLICATION

SESSION
MANAGER

ICCCM-compliant Communication

X Protocol
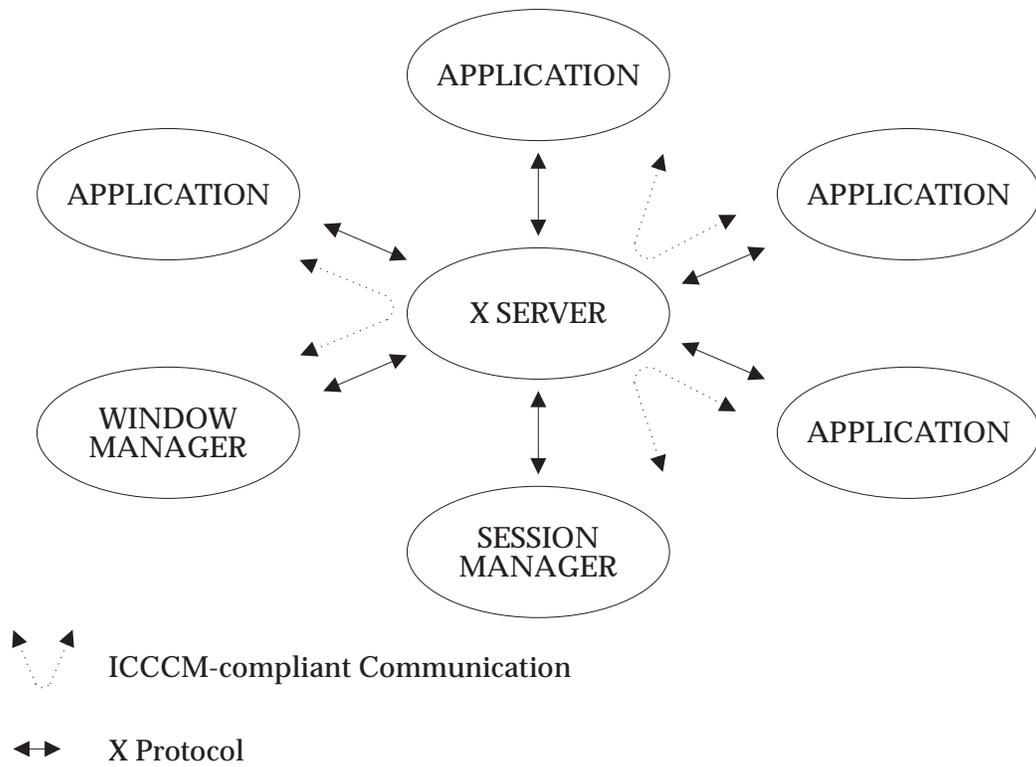
Figure 1-**5** X Application Relationships

Introduction to Xlib

The X Window System is a network-transparent window system that was designed at MIT. X display servers run on computers with either monochrome or color bitmap display hardware. The server distributes user input to and accepts output requests from various client programs located either on the same machine or elsewhere in the network. Xlib is a C subroutine library that application programs (clients) use to interface with the window system by means of a stream connection. Although a client usually runs on the same machine as the X server it is talking to, this need not be the case.

This document is a reference guide to the low-level C-language interface to the *X Window System Protocol*. It is neither a tutorial nor a user's guide to programming the X Window System. Rather, it provides a detailed description of each function in the library as well as a discussion of the related background information. This document assumes a basic understanding of a graphics window system and of the C programming language. Other higher-level abstractions (for example, those provided by the toolkits for X) are built on top of the Xlib library. For further information about these higher-level libraries, see the **Toolkit Intrinsics** specification. The X Window System Protocol provides the definitive word on the behaviour of X. Although additional information appears here, the protocol document is the ruling document.

To provide an introduction to X programming, this chapter discusses:

- overview of the X Window System
- errors
- standard header files
- naming and argument conventions
- programming considerations
- formatting conventions.

2.1   General Concepts

Refer to Chapter 1 for an introduction to the concepts and terminology of X Windows. A child window can be larger than its parent. That is, part or all of the child window can extend beyond the boundaries of the parent, but all output to a window is clipped by its parent. If several children of a window have overlapping locations, one of the children is considered to be on top of or raised over the others thus obscuring them. Output to areas covered by other windows is suppressed by the window system unless the window has backing store. If a window is obscured by a second window, the second window obscures only those ancestors of the second window, which are also ancestors of the first window.

A window has a border zero or more pixels in width, which can be any pattern (pixmap) or solid color you like. A window usually, but not always, has a background pattern, which will be repainted by the window system when uncovered. Child windows obscure their parents, and graphic operations in the parent window usually are clipped by the children.

Each window and pixmap has its own coordinate system. The coordinate system has the X axis horizontal and the Y axis vertical, with the origin [0, 0] at the upper left. Coordinates are integral, in terms of pixels, and coincide with pixel centers. For a window, the origin is inside the border at the inside upper left.

X does not guarantee to preserve the contents of windows. When part or all of a window is hidden and then brought back onto the screen, its contents may be lost. The server then sends the client program an *Expose* event to notify it that part or all of the window needs to be repainted. Programs must be prepared to regenerate the contents of windows on demand.

X also provides off-screen storage of graphics objects, called pixmaps. Single plane (depth 1) pixmaps are sometimes referred to as bitmaps. Pixmaps can be used in most graphics functions interchangeably with windows and are used in various graphics operations to define patterns or tiles. Windows and pixmaps together are referred to as drawables.

Most of the functions in Xlib just add requests to an output buffer. These requests later execute asynchronously on the X server. Functions that return values of information stored in the server do not return (that is, they block) until an explicit reply is received or an error occurs. You can provide an error handler, which will be called when the error is reported.

If a client does not want a request to execute asynchronously, it can follow the request with a call to *XSync*, which blocks until all previously buffered asynchronous events have been sent and acted on. As an important side effect, the output buffer in Xlib is always flushed by a call to any function that returns a value from the server or waits for input.

Many Xlib functions will return an integer resource ID, which allows you to refer to objects stored on the X server. These can be of type *Window*, *Font*, *Pixmap*, *Colormap*, *Cursor* and *GContext*, as defined in the file <**X11/X.h**>. These resources are created by requests and are destroyed (or freed) by requests or when connections are closed. Most of these resources are potentially sharable between applications, and in fact, windows are manipulated explicitly by window manager programs. Fonts and cursors are shared automatically across multiple screens. Fonts are loaded and unloaded as needed and are shared by multiple clients. Fonts are often cached in the server. Xlib provides no support for sharing graphics contexts between applications.

Client programs are informed of events. Events may either be side effects of a request (for example, restacking windows generates *Expose* events) or completely asynchronous (for example, from the keyboard). A client program asks to be informed of events. Because other applications can send events to your application, programs must be prepared to handle (or ignore) events of all types.

Input events (for example, a key pressed or the pointer moved) arrive asynchronously from the server and are queued until they are requested by an explicit call (for example, *XNextEvent* or *XWindowEvent*). In addition, some library functions (for example, *XRaiseWindow*) generate *Expose* and *ConfigureRequest* events. These events also arrive asynchronously, but the client may wish to explicitly wait for them by calling *XSync* after calling a function that can cause the server to generate events.

2.2   Status

The Xlib programmatic interface defined by this document is syntactically correct and complete. All the function definitions specified in this document are mandatory, including the protocol extension mechanism.

Although the X Window System provides a protocol extension mechanism to allow for additional features, these additional features are not standardized and often will be vendor-specific. Their use, although useful and necessary in some applications, cannot be guaranteed to be present in all computers which implement the X Window System.

No extensions are described here.

Internationalization

The X Window System is 8-bit transparent. Any 8-bit or 16-bit codeset may be used in the font and text calls. In addition, 8-bit codesets may be used in all strings including filenames, atom names and color names.

Error messages in an X/Open conformant X Window System implementation will be internationalizable in order to support localization. For more information, see the **X/Open Internationalisation Guide**.

2.3   Errors

Some functions return *Status*, an integer error indication. If the function fails, it returns a zero. If the function returns a status of zero, it has not updated the return arguments. Because C does not provide multiple return values, many functions must return their results by writing into client-passed storage. By default, errors are handled either by a standard library function or by one that you provide. Functions that return pointers to strings return NULL pointers if the string does not exist.

The X server reports protocol errors at the time that it detects them. If more than one error could be generated for a given request, the server can report any of them.

Because Xlib usually does not transmit requests to the server immediately (that is, it buffers them), errors can be reported much later than they actually occur. For debugging purposes, however, Xlib provides a mechanism for forcing synchronous behaviour (see Section 12.8.1). When synchronization is enabled, errors are reported as they are generated.

When Xlib detects an error, it calls an error handler, which your program can provide. If you do not provide an error handler, the error is printed, and your program terminates.

2.4   Standard Header Files

The following include files are part of the Xlib standard.

<X11/Xlib.h>
> This is the main header file for Xlib. The majority of all Xlib symbols are declared by including this file. This file also contains the preprocessor symbol *XlibSpecificationRelease*. This symbol is defined to have the ''5'' in this release of the standard. (Earlier releases of Xlib did not have this symbol.)

<X11/X.h>
> This file declares types and constants for the X protocol that are to be used by applications. It is included automatically from **<X11/Xlib.h>**, so application code should never need to reference this file directly.

<X11/Xcms.h>
> This file contains symbols for much of the color management facilities described in Chapter 7. All functions, types and symbols with the prefix ''Xcms'', plus the Color Conversion Contexts macros, are declared in this file. **<X11/Xlib.h>** must be included before including this file.

<X11/Xutil.h>
> This file declares various functions, types and symbols used for inter-client communication and application utility functions, described in Chapter 15 and Chapter 17. **<X11/Xlib.h>** must be included before including this file.

<X11/Xresource.h>
> This file declares all functions, types, and symbols for the resource manager facilities, described in Chapter 16. **<X11/Xlib.h>** must be included before including this file.

<X11/Xatom.h>
> This file declares all predefined atoms, symbols with prefix ''XA_''.

<X11/cursorfont.h>
> This file declares the cursor symbols for the standard cursor font, listed in Appendix B. All symbols have the prefix ''XC_''.

<X11/keysymdef.h>
> This file declares all standard KeySym values, symbols with prefix ''XK_''. The KeySyms are arranged in groups, and a preprocessor symbol controls inclusion of each group. The preprocessor symbol must be defined prior to inclusion of the file to obtain the associated values. The preprocessor symbols are: XK_MISCELLANY, XK_LATIN1, XK_LATIN2, XK_LATIN3, XK_LATIN4, XK_KATAKANA, XK_ARABIC, XK_CYRILLIC, XK_GREEK, XK_TECHNICAL, XK_SPECIAL, XK_PUBLISHING, XK_APL and XK_HEBREW.

<X11/keysym.h>
> This file defines the preprocessor symbols XK_MISCELLANY, XK_LATIN1, XK_LATIN2, XK_LATIN3, XK_LATIN4 and XK_GREEK, and then includes **<X11/keysymdef.h>**.

<X11/Xlibint.h>
> This file declares all the functions, types and symbols used for extensions, described in Appendix B. This file automatically includes **<X11/Xlib.h>**.

<X11/Xproto.h>
> This file declares types and symbols for the basic X protocol, for use in implementing extensions. It is included automatically from **<X11/Xlibint.h>**, so application and extension code should never need to reference this file directly.

<X11/Xprotostr.h>

This file declares types and symbols for the basic X protocol, for use in implementing extensions. It is included automatically from **<X11/Xproto.h>**, so application and extension code should never need to reference this file directly.

<X11/X10.h>

This file declares all the functions, types and symbols used for the X10 compatibility functions, described in Appendix D.

2.5 Generic Values and Types

The following symbols are defined by Xlib and used throughout the manual:

- Xlib defines the type *Bool* and the boolean values *True* and *False*.

- *None* is the universal null resource ID or atom.

- The type *XID* is used for generic resource IDs.

- The type *XPointer* is defined to be ''char *'' and is used as a generic opaque pointer to data.

2.6   Naming and Argument Conventions within Xlib

Xlib follows a number of conventions for the naming and syntax of the functions. Given that you remember what information the function requires, these conventions are intended to make the syntax of the functions more predictable.

The major naming conventions are:

- To differentiate the X symbols from the other symbols, the library uses mixed case for external symbols. It leaves lower case for variables and all upper case for user macros, as per existing convention.

- All Xlib functions begin with a capital X.

- The beginnings of all function names and symbols are capitalized.

- All user-visible data structures begin with a capital X. More generally, anything that a user might dereference begins with a capital X.

- Macros and other symbols do not begin with a capital X. To distinguish them from all user symbols, each word in the macro is capitalized.

- All elements of or variables in a data structure are in lower case. Compound words, where needed, are constructed with underscores (_).

- The display argument, where used, is always first in the argument list.

- All resource objects, where used, occur at the beginning of the argument list immediately after the display argument.

- When a graphics context is present together with another type of resource (most commonly, a drawable), the graphics context occurs in the argument list after the other resource. Drawables outrank all other resources.

- Source arguments always precede the destination arguments in the argument list.

- The x argument always precedes the y argument in the argument list.

- The width argument always precedes the height argument in the argument list.

- Where the x, y, width and height arguments are used together, the x and y arguments always precede the width and height arguments.

- Where a mask is accompanied with a structure, the mask always precedes the pointer to the structure in the argument list.

2.7   **Programming Considerations**

The major programming considerations are:

- Coordinates and sizes in X are actually 16-bit quantities. This decision was taken to minimize the bandwidth required for a given level of performance. Coordinates usually are declared as an ''int'' in the interface. Values larger than 16 bits are truncated silently. Sizes (width and height) are declared as unsigned quantities.

- Keyboards are the greatest variable between different manufacturers' workstations. If you want your program to be portable, you should be particularly conservative here.

- Many display systems have limited amounts of off-screen memory. If you can, you should minimize use of pixmaps and backing store.

- The user should have control of his screen real estate. Therefore, you should write your applications to react to window management rather than presume control of the entire screen. What you do inside of your top-level window, however, is up to your application. For further information, see Chapter 15 and the **ICCCM** specification.

2.8   Character Sets and Encodings

Some of the Xlib functions make reference to specific character sets and character encodings. The following ones are the most common:

X Portable Character Set

A basic set of 97 characters which are assumed to exist in all locales supported by Xlib. This set contains the following characters:

```
a..z A..Z 0..9
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
<space>, <tab> and <newline>
```

This is the left/lower half of the graphic character set of ISO 8859-1 plus <space>, <tab> and <newline>. It is also the set of graphic characters in 7-bit ASCII plus the same three control characters. The actual encoding of these characters on the host is system-dependent.

Host Portable Character Encoding

The encoding of the X Portable Character Set on the host. The encoding itself is not defined by this standard, but the encoding must be the same in all locales supported by Xlib on the host. If a string is said to be in the Host Portable Character Encoding, then it only contains characters from the X Portable Character Set, in the host encoding.

Latin-1

The coded character set defined by ISO 8859-1.

STRING encoding

Latin-1, plus tab and newline.

POSIX Portable Filename Character Set

The set of 65 characters which can be used in naming files on a POSIX-compliant host that are correctly processed in all locales. The set is:

```
a..z A..Z 0..9 ._-
```

2.9   **Formatting Conventions**

This document uses the following conventions:

- Global symbols are printed in *italics*. These can be either function names, symbols defined in include files, or structure names. Arguments are printed in *italics*.

- Each function is introduced by a general discussion that distinguishes it from other functions. The function declaration itself follows, and each argument is specifically explained. Although ANSI C function prototype syntax is not used, Xlib header files normally declare functions using function prototypes in ANSI C environments. General discussion of the function, if any is required, follows the arguments. Where applicable, the last paragraph of the explanation lists the possible Xlib error codes that the function can generate. For a complete discussion of the Xlib error codes, see Section 12.8.2 on page 258.

- To eliminate any ambiguity between those arguments that you pass and those that a function returns to you, the explanations for all arguments that you pass start with the word *specifies* or, in the case of multiple arguments, the word *specify*. The explanations for all arguments that are returned to you start with the word *returns* or, in the case of multiple arguments, the word *return*. The explanations for all arguments that you can pass and are returned start with the words *specifies and returns*.

- Any pointer to a structure that is used to return a value is designated as such by the *_return* suffix as part of its name. All other pointers passed to these functions are used for reading only. A few arguments use pointers to structures that are used for both input and output and are indicated by using the *_in_out* suffix.

Display Functions

Before your program can use a display, you must establish a connection to the X server. Once you have established a connection, you then can use the Xlib macros and functions discussed in this chapter to return information about the display. This chapter discusses how to:

- open (connect to) the display

- obtain information about the display, image format and screen

- free client-created data

- close (disconnect from) a display.

The chapter concludes with a general discussion of what occurs when the connection to the X server is closed.

3.1   Opening the Display

To open a connection to the X server that controls a display, use *XOpenDisplay*.

```
Display *XOpenDisplay(display_name)
      char *display_name;
```

display_name   Specifies the hardware display name, which determines the display and communications domain to be used. On a POSIX-conformant system, if the display_name is NULL, it defaults to the value of the DISPLAY environment variable.

The encoding and interpretation of the display name is implementation-dependent. Strings in the Host Portable Character Encoding are supported; support for other characters is implementation-dependent. On POSIX-conformant systems, the display name or DISPLAY environment variable can be a string in the format:

```
hostname:number.screen_number
```

hostname   Specifies the name of the host machine on which the display is physically attached. You follow the hostname with either a single colon (:) or a double colon (::).

number   Specifies the number of the display server on that host machine. You may optionally follow this display number with a period (.). A single CPU can have more than one display. Multiple displays are usually numbered starting with zero.

screen_number   Specifies the screen to be used on that server. Multiple screens can be controlled by a single X server. The *screen_number* sets an internal variable that can be accessed by using the *DefaultScreen* macro or the *XDefaultScreen* function if you are using languages other than C (see Section 3.2.1).

For example, the following would specify screen 1 of display 0 on the machine named ''dual-headed'':

```
dual-headed:0.1
```

The *XOpenDisplay* function returns a *Display* structure that serves as the connection to the X server and that contains all the information about that X server. *XOpenDisplay* connects your application to the X server through TCP or DECnet communications protocols, or through some local inter-process communication protocol. If the hostname is a host machine name and a single colon (:) separates the hostname and display number, *XOpenDisplay* connects using TCP streams. If the hostname is not specified, Xlib uses whatever it believes is the fastest transport. If the hostname is a host machine name and a double colon (::) separates the hostname and display number, *XOpenDisplay* connects using DECnet. A single X server can support any or all of these transport mechanisms simultaneously. A particular Xlib implementation can support many more of these transport mechanisms.

If successful, *XOpenDisplay* returns a pointer to a *Display* structure, which is defined in <**X11/Xlib.h**>. If *XOpenDisplay* does not succeed, it returns NULL. After a successful call to *XOpenDisplay*, all of the screens in the display can be used by the client. The screen number specified in the display_name argument is returned by the *DefaultScreen* macro (or the *XDefaultScreen* function). You can access elements of the *Display* and *Screen* structures only by using the information macros or functions. For information about using macros and functions to obtain information from the *Display* structure, see Section 3.2.1.

X servers may implement various types of access control mechanisms (see Section 10.8).

Stamp:XXXXXXXXXXXXXXXXXXXXXXXXX

3.2   Obtaining Information about the Display, Image Formats or Screens

The Xlib library provides a number of useful macros and corresponding functions that return data from the *Display* structure. The macros are used for C programming, and their corresponding function equivalents are for other language bindings. This section discusses:

- display macros

- image format macros

- screen macros.

All other members of the *Display* structure (that is, those for which no macros are defined) are private to Xlib and must not be used. Applications must never directly modify or inspect these private members of the *Display* structure.

Note: The *XDisplayWidth*, *XDisplayHeight*, *XDisplayCells*, *XDisplayPlanes*, *XDisplayWidthMM* and *XDisplayHeightMM* functions in the next sections are misnamed. These functions really should be named Screen*whatever* and XScreen*whatever*, not Display*whatever* or XDisplay*whatever*. Our apologies for the resulting confusion.

3.2.1   Display Macros

Applications should not directly modify any part of the *Display* and *Screen* structures. The members should be considered read-only, although they may change as the result of other operations on the display.

The following lists the C-language macros, their corresponding function equivalents that are for other language bindings, and what data they both can return.

```
AllPlanes
```
```
unsigned long XAllPlanes()
```

Both return a value with all bits set to 1 suitable for use in a plane argument to a procedure.

Both *BlackPixel* and *WhitePixel* can be used in implementing a monochrome application. These pixel values are for permanently allocated entries in the default colormap. The actual RGB (red, green and blue) values are settable on some screens and, in any case, may not actually be black or white. The names are intended to convey the expected relative intensity of the colors.

```
BlackPixel(display, screen_number)
```
```
unsigned long XBlackPixel(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the black pixel value for the specified screen.

```
WhitePixel(display, screen_number)
```
```
unsigned long XWhitePixel(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the white pixel value for the specified screen.

```
ConnectionNumber(display)
```

```
int XConnectionNumber(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return a connection number for the specified display. On a POSIX-conformant system, this is the file descriptor of the connection.

```
DefaultColormap(display, screen_number)
```

```
Colormap XDefaultColormap(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the default colormap ID for allocation on the specified screen. Most routine allocations of color should be made out of this colormap.

```
DefaultDepth(display, screen_number)
```

```
int XDefaultDepth(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the depth (number of planes) of the default root window for the specified screen. Other depths may also be supported on this screen (see *XMatchVisualInfo*).

To determine the number of depths that are available on a given screen, use *XListDepths*.

```
int *XListDepths(display, screen_number, count_return)
      Display *display;
      int screen_number;
      int *count_return;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

count_return   Returns the number of depths.

The *XListDepths* function returns the array of depths that are available on the specified screen. If the specified *screen_number* is valid and sufficient memory for the array can be allocated, *XListDepths* sets *count_return* to the number of available depths. Otherwise, it does not set *count_return* and returns NULL. To release the memory allocated for the array of depths, use *XFree*.

```
DefaultGC(display, screen_number)
```

```
GC XDefaultGC(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the default graphics context for the root window of the specified screen. This GC is created for the convenience of simple applications and contains the default GC components with the foreground and background pixel values initialized to the black and white pixels for the screen, respectively. You can modify its contents freely because it is not used in any Xlib function. This GC should never be freed.

```
DefaultRootWindow(display)
```

```
Window XDefaultRootWindow(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return the root window for the default screen.

```
DefaultScreenOfDisplay(display)
```

```
Screen *XDefaultScreenOfDisplay(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return a pointer to the default screen.

```
ScreenOfDisplay(display, screen_number)
```

```
Screen *XScreenOfDisplay(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return a pointer to the indicated screen.

```
DefaultScreen(display)
```

```
int XDefaultScreen(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return the default screen number referenced by the *XOpenDisplay* function. This macro or function should be used to retrieve the screen number in applications that will use only a single screen.

```
DefaultVisual(display, screen_number)

Visual *XDefaultVisual(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the default visual type for the specified screen. For further information about visual types, see Section 4.1.

```
DisplayCells(display, screen_number)

int XDisplayCells(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the number of entries in the default colormap.

```
DisplayPlanes(display, screen_number)

int XDisplayPlanes(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the depth of the root window of the specified screen. For an explanation of depth, see the glossary.

```
DisplayString(display)

char *XDisplayString(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return the string that was passed to *XOpenDisplay* when the current display was opened. On POSIX-conformant systems, if the passed string was NULL, these return the value of the DISPLAY environment variable when the current display was opened. These are useful to applications that invoke the *fork* system call and want to open a new connection to the same display from the child process as well as for printing error messages.

```
long XMaxRequestSize(display)
      Display *display;
```

display   Specifies the connection to the X server.

XMaxRequestSize returns the maximum request size (in 4-byte units) supported by the server. Single protocol requests to the server can be no longer than this size. The protocol guarantees the size to be no smaller than 4096 units (16384 bytes). Xlib automatically breaks data up into multiple protocol requests as necessary for the following functions: *XDrawPoints*, *XDrawRectangles*, *XDrawSegments*, *XFillArcs*, *XFillRectangles* and *XPutImage*.

```
LastKnownRequestProcessed(display)
```

```
unsigned long XLastKnownRequestProcessed(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both extract the full serial number of the last request known by Xlib to have been processed by the X server. Xlib automatically sets this number when replies, events and errors are received.

```
NextRequest(display)
```

```
unsigned long XNextRequest(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both extract the full serial number that is to be used for the next request. Serial numbers are maintained separately for each display connection.

```
ProtocolVersion(display)
```

```
int XProtocolVersion(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return the major version number (11) of the X protocol associated with the connected display.

```
ProtocolRevision(display)
```

```
int XProtocolRevision(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return the minor protocol revision number of the X server.

```
QLength(display)
```

```
int XQLength(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return the length of the event queue for the connected display. Note that there may be more events that have not been read into the queue yet (see *XEventsQueued*).

```
RootWindow(display, screen_number)
```

```
Window XRootWindow(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the root window. These are useful with functions that need a drawable of a particular screen and for creating top-level windows.

```
ScreenCount(display)
```

```
int XScreenCount(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return the number of available screens.

```
ServerVendor(display)
```

```
char *XServerVendor(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return a pointer to a null-terminated string that provides some identification of the owner of the X server implementation. If the data returned by the server is in the Latin Portable Character Encoding, then the string is in the Host Portable Character Encoding. Otherwise, the contents of the string are implementation-dependent.

```
VendorRelease(display)
```

```
int XVendorRelease(display)
      Display *display;
```

display   Specifies the connection to the X server.

Both return a number related to a vendor's release of the X server.

3.2.2   Image Format Functions and Macros

Applications are required to present data to the X server in a format that the server demands. To help simplify applications, most of the work required to convert the data is provided by Xlib (see Section 9.7 and Section 17.8).

The *XPixmapFormatValues* structure provides an interface to the pixmap format information that is returned at the time of a connection setup. It contains:

```
typedef struct {
      int depth;
      int bits_per_pixel;
      int scanline_pad;
} XPixmapFormatValues;
```

To obtain the pixmap format information for a given display, use *XListPixmapFormats*.

```
XPixmapFormatValues *XListPixmapFormats(display, count_return)
      Display *display;
      int *count_return;
```

display   Specifies the connection to the X server.

count_return   Returns the number of pixmap formats that are supported by the display.

The *XListPixmapFormats* function returns an array of *XPixmapFormatValues* structures that describe the types of Z format images supported by the specified display. If insufficient memory is available, *XListPixmapFormats* returns NULL. To free the allocated storage for the *XPixmapFormatValues* structures, use *XFree*.

The following lists the C language macros, their corresponding function equivalents that are for other language bindings, and what data they both return for the specified server and screen. These are often used by toolkits as well as by simple applications.

```
ImageByteOrder(display)
```

```
int XImageByteOrder(display)
    Display *display;
```

display   Specifies the connection to the X server.

Both specify the required byte order for images for each scanline unit in XY format (bitmap) or for each pixel value in Z format. The macro or function can return either *LSBFirst* or *MSBFirst*.

```
BitmapUnit(display)
```

```
int XBitmapUnit(display)
    Display *display;
```

display   Specifies the connection to the X server.

Both return the size of a bitmap's scanline unit in bits. The scanline is calculated in multiples of this value.

```
BitmapBitOrder(display)
```

```
int XBitmapBitOrder(display)
    Display *display;
```

display   Specifies the connection to the X server.

Within each bitmap unit, the left-most bit in the bitmap as displayed on the screen is either the least-significant or most-significant bit in the unit. This macro or function can return *LSBFirst* or *MSBFirst*.

```
BitmapPad(display)
```

```
int XBitmapPad(display)
    Display *display;
```

display   Specifies the connection to the X server.

Each scanline must be padded to a multiple of bits returned by this macro or function.

```
DisplayHeight(display, screen_number)
```

```
int XDisplayHeight(display, screen_number)
    Display *display;
    int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return an integer that describes the height of the screen in pixels.

```
DisplayHeightMM(display, screen_number)

int XDisplayHeightMM(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the height of the specified screen in millimeters.

```
DisplayWidth(display, screen_number)

int XDisplayWidth(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the width of the screen in pixels.

```
DisplayWidthMM(display, screen_number)

int XDisplayWidthMM(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

Both return the width of the specified screen in millimeters.

3.2.3   Screen Information Macros

The following lists the C-language macros, their corresponding function equivalents that are for other language bindings, and what data they both can return. These macros or functions all take a pointer to the appropriate screen structure.

```
BlackPixelOfScreen(screen)

unsigned long XBlackPixelOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the black pixel value of the specified screen.

```
WhitePixelOfScreen(screen)

unsigned long XWhitePixelOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the white pixel value of the specified screen.

```
CellsOfScreen(screen)
```

```
int XCellsOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the number of colormap cells in the default colormap of the specified screen.

```
DefaultColormapOfScreen(screen)
```

```
Colormap XDefaultColormapOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the default colormap of the specified screen.

```
DefaultDepthOfScreen(screen)
```

```
int XDefaultDepthOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the depth of the root window.

```
DefaultGCOfScreen(screen)
```

```
GC XDefaultGCOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return a default graphics context (GC) of the specified screen, which has the same depth as the root window of the screen. The GC must never be freed.

```
DefaultVisualOfScreen(screen)
```

```
Visual *XDefaultVisualOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the default visual of the specified screen. For information on visual types, see Section 4.1.

```
DoesBackingStore(screen)
```

```
int XDoesBackingStore(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return a value indicating whether the screen supports backing stores. The value returned can be one of *WhenMapped*, *NotUseful* or *Always* (see Section 4.2.4).

```
DoesSaveUnders(screen)
```

```
Bool XDoesSaveUnders(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return a Boolean value indicating whether the screen supports save unders. If *True*, the screen supports save unders. If *False*, the screen does not support save unders (see Section 4.2.5).

```
DisplayOfScreen(screen)
```

```
Display *XDisplayOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the display of the specified screen.

```
int XScreenNumberOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

The *XScreenNumberOfScreen* function returns the screen index number of the specified screen.

```
EventMaskOfScreen(screen)
```

```
long XEventMaskOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the event mask of the root window for the specified screen at connection setup time.

```
WidthOfScreen(screen)
```

```
int XWidthOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the width of the specified screen in pixels.

```
HeightOfScreen(screen)
```

```
int XHeightOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the height of the specified screen in pixels.

```
WidthMMOfScreen(screen)
```

```
int XWidthMMOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the width of the specified screen in millimeters.

```
HeightMMOfScreen(screen)
```

```
int XHeightMMOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the height of the specified screen in millimeters.

```
MaxCmapsOfScreen(screen)
```

```
int XMaxCmapsOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the maximum number of installed colormaps supported by the specified screen (see Section 10.3).

```
MinCmapsOfScreen(screen)
```

```
int XMinCmapsOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the minimum number of installed colormaps supported by the specified screen (see Section 10.3).

```
PlanesOfScreen(screen)
```

```
int XPlanesOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the depth of the root window.

```
RootWindowOfScreen(screen)
```

```
Window XRootWindowOfScreen(screen)
      Screen *screen;
```

screen   Specifies the appropriate *Screen* structure.

Both return the root window of the specified screen.

3.3   Generating a NoOperation Protocol Request

To execute a *NoOperation* protocol request, use *XNoOp*.

```
XNoOp(display)
        Display *display;
```

display   Specifies the connection to the X server.

The *XNoOp* function sends a *NoOperation* protocol request to the X server, thereby exercising the connection.

3.4   Freeing Client-created Data

To free in-memory data that was created by an Xlib function, use *XFree*.

```
XFree(data)
       void *data;
```

data   Specifies the data that is to be freed.

The *XFree* function is a general-purpose Xlib routine that frees the specified data. You must use it to free any objects that were allocated by Xlib, unless an alternate function is explicitly specified for the object.

3.5   Closing the Display

To close a display or disconnect from the X server, use *XCloseDisplay*.

```
XCloseDisplay(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XCloseDisplay* function closes the connection to the X server for the display specified in the *Display* structure and destroys all windows, resource IDs (*Window*, *Font*, *Pixmap*, *Colormap*, *Cursor* and *GContext*), or other resources that the client has created on this display, unless the close-down mode of the resource has been changed (see *XSetCloseDownMode*). Therefore, these windows, resource IDs and other resources should never be referenced again or an error will be generated. Before exiting, you should call *XCloseDisplay* explicitly so that any pending errors are reported as *XCloseDisplay* performs a final *XSync* operation.

XCloseDisplay can generate a *BadGC* error.

Xlib provides a function to permit the resources owned by a client to survive after the client's connection is closed. To change a client's close-down mode, use *XSetCloseDownMode*.

```
XSetCloseDownMode(display, close_mode)
      Display *display;
      int close_mode;
```

display   Specifies the connection to the X server.

close_mode   Specifies   the   client   close-down   mode.   You   can   pass   *DestroyAll*,
                     RetainPermanent or *RetainTemporary*.

The *XSetCloseDownMode* defines what will happen to the client's resources at connection close. A connection starts in *DestroyAll* mode. For information on what happens to the client's resources when the close_mode argument is *RetainPermanent* or *RetainTemporary*, see Section 3.6.

XSetCloseDownMode can generate a *BadValue* error.

3.6   X Server Connection Close Operations

When the X server's connection to a client is closed either by an explicit call to *XCloseDisplay* or by a process that exits, the X server performs the following automatic operations:

- It disowns all selections owned by the client (see *XSetSelectionOwner*).

- It performs an *XUngrabPointer* and *XUngrabKeyboard* if the client has actively grabbed the pointer or the keyboard.

- It performs an *XUngrabServer* if the client has grabbed the server.

- It releases all passive grabs made by the client.

- It marks all resources (including colormap entries) allocated by the client either as permanent or temporary, depending on whether the close-down mode is *RetainPermanent* or *RetainTemporary*. However, this does not prevent other client applications from explicitly destroying the resources (see *XSetCloseDownMode*).

When the close-down mode is *DestroyAll*, the X server destroys all of a client's resources as follows:

- It examines each window in the client's save-set to determine if it is an inferior (subwindow) of a window created by the client. (The save-set is a list of other clients' windows, which are referred to as save-set windows.) If so, the X server reparents the save-set window to the closest ancestor so that the save-set window is not an inferior of a window created by the client. The reparenting leaves unchanged the absolute coordinates (with respect to the root window) of the upper-left outer corner of the save-set window.

- It performs a *MapWindow* request on the save-set window if the save-set window is unmapped. The X server does this even if the save-set window was not an inferior of a window created by the client.

- It destroys all windows created by the client.

- It performs the appropriate free request on each non-window resource created by the client in the server (for example, *Font*, *Pixmap*, *Cursor*, *Colormap* and *GContext*).

- It frees all colors and colormap entries allocated by a client application.

Additional processing occurs when the last connection to the X server closes. An X server goes through a cycle of having no connections and having some connections. When the last connection to the X server closes as a result of a connection closing with the close_mode of *DestroyAll*, the X server does the following:

- It resets its state as if it had just been started. The X server begins by destroying all lingering resources from clients that have terminated in *RetainPermanent* or *RetainTemporary* mode.

- It deletes all but the predefined atom identifiers.

- It deletes all properties on all root windows (see Section 5.3).

- It resets all device maps and attributes (for example, key click, bell volume and acceleration) as well as the access control list.

- It restores the standard root tiles and cursors.

- It restores the default font path.

- It restores the input focus to state *PointerRoot*.

However, the X server does not reset if you close a connection with a close-down mode set to *RetainPermanent* or *RetainTemporary*.

Window Functions

In the X Window System, a window is a rectangular area on the screen that lets you view graphic output. Client applications can display overlapping and nested windows on one or more screens that are driven by X servers on one or more machines. Clients who want to create windows must first connect their program to the X server by calling *XOpenDisplay*. This chapter begins with a discussion of visual types and window attributes. The chapter continues with a discussion of the Xlib functions you can use to:

- create windows
- destroy windows
- map windows
- unmap windows
- configure windows
- change the stacking order
- change window attributes.

This chapter also identifies the window actions that may generate events.

Note that it is vital that your application conform to the established conventions for communicating with window managers for it to work well with the various window managers in use (see Section 15.1). Toolkits generally adhere to these conventions for you, relieving you of the burden. Toolkits also often supersede many functions in this chapter with versions of their own. Refer to the documentation for the toolkit you are using for more information.

4.1   Visual Types

On some display hardware, it may be possible to deal with color resources in more than one way. For example, you may be able to deal with a screen of either 12-bit depth with arbitrary mapping of pixel to color (pseudo-color) or 24-bit depth with 8 bits of the pixel dedicated to each of red, green and blue. These different ways of dealing with the visual aspects of the screen are called visuals. For each screen of the display, there may be a list of valid visual types supported at different depths of the screen. Because default windows and visual types are defined for each screen, most simple applications need not deal with this complexity. Xlib provides macros and functions that return the default root window, the default depth of the default root window, and the default visual type (see Section 3.2.1 and Section 17.7).

Xlib uses an opaque *Visual* structure that contains information about the possible color mapping. The visual utility functions (see Section 17.7) use an *XVisualInfo* structure to return this information to an application. The members of this structure pertinent to this discussion are class, red_mask, green_mask, blue_mask, bits_per_rgb and colormap_size. The class member specifies one of the possible visual classes of the screen and can be *StaticGray*, *StaticColor*, *TrueColor*, *GrayScale*, *PseudoColor* or *DirectColor*.

The following concepts may serve to make the explanation of visual types clearer. The screen can be color or grayscale, can have a colormap that is writable or read-only, and can also have a colormap whose indices are decomposed into separate RGB pieces, provided one is not on a grayscale screen. This leads to the following diagram:

	Color		GrayScale	
	R/O	R/W	R/O	R/W
Undecomposed Colormap	Static Color	Pseudo Color	Static Gray	Gray Scale
Decomposed Colormap	True Color	Direct Color		

Conceptually, as each pixel is read out of video memory for display on the screen, it goes through a look-up stage by indexing into a colormap. Colormaps can be manipulated arbitrarily on some hardware, in limited ways on other hardware, and not at all on other hardware. The visual types affect the colormap and the RGB values in the following ways:

- For *PseudoColor*, a pixel value indexes a colormap to produce independent RGB values, and the RGB values can be changed dynamically.

- *GrayScale* is treated the same way as *PseudoColor* except that the primary that drives the screen is undefined. Thus, the client should always store the same value for red, green and blue in the colormaps.

- For *DirectColor*, a pixel value is decomposed into separate RGB subfields, and each subfield separately indexes the colormap for the corresponding value. The RGB values can be changed dynamically.

- *TrueColor* is treated the same way as *DirectColor* except that the colormap has predefined, read-only RGB values. These RGB values are server-dependent but provide linear or near-linear ramps in each primary.

- *StaticColor* is treated the same way as *PseudoColor* except that the colormap has predefined, read-only, server-dependent RGB values.

- *StaticGray* is treated the same way as *StaticColor* except that the RGB values are equal for any single pixel value, thus resulting in shades of gray. *StaticGray* with a two-entry colormap can be thought of as monochrome.

The red_mask, green_mask and blue_mask members are only defined for *DirectColor* and *TrueColor*. Each has one contiguous set of bits with no intersections. The bits_per_rgb member specifies the log base 2 of the number of distinct color values (individually) of red, green and blue. Actual RGB values are unsigned 16-bit numbers. The colormap_size member defines the number of available colormap entries in a newly created colormap. For *DirectColor* and *TrueColor*, this is the size of an individual pixel subfield.

To obtain the visual ID from a *Visual*, use *XVisualIDFromVisual*.

```
VisualID XVisualIDFromVisual(visual)
        Visual *visual;
```

visual   Specifies the visual type.

The *XVisualIDFromVisual* function returns the visual ID for the specified visual type.

4.2   **Window Attributes**

All *InputOutput* windows have a border width of zero or more pixels, an optional background, an event suppression mask (which suppresses propagation of events from children), and a property list (see Section 5.3). The window border and background can be a solid color or a pattern, called a tile. All windows except the root have a parent and are clipped by their parent. If a window is stacked on top of another window, it obscures that other window for the purpose of input. If a window has a background (almost all do), it obscures the other window for purposes of output. Attempts to output to the obscured area do nothing, and no input events (for example, pointer motion) are generated for the obscured area.

Windows also have associated property lists (see Section 5.3).

Both *InputOutput* and *InputOnly* windows have the following common attributes, which are the only attributes of an *InputOnly* window:

- win-gravity

- event-mask

- do-not-propagate-mask

- override-redirect

- cursor.

If you specify any other attributes for an *InputOnly* window, a *BadMatch* error results.

InputOnly windows are used for controlling input events in situations where *InputOutput* windows are unnecessary. *InputOnly* windows are invisible; can only be used to control such things as cursors, input event generation and grabbing; and cannot be used in any graphics requests. Note that *InputOnly* windows cannot have *InputOutput* windows as inferiors.

Windows have borders of a programmable width and pattern as well as a background pattern or tile. Pixel values can be used for solid colors. The background and border pixmaps can be destroyed immediately after creating the window if no further explicit references to them are to be made. The pattern can either be relative to the parent or absolute. If *ParentRelative*, the parent's background is used.

When windows are first created, they are not visible (not mapped) on the screen. Any output to a window that is not visible on the screen and that does not have backing store will be discarded. An application may wish to create a window long before it is mapped to the screen. When a window is eventually mapped to the screen (using *XMapWindow*), the X server generates an *Expose* event for the window if backing store has not been maintained.

A window manager can override your choice of size, border width and position for a top-level window. Your program must be prepared to use the actual size and position of the top window. It is not acceptable for a client application to resize itself unless in direct response to a human command to do so. Instead, either your program should use the space given to it, or if the space is too small for any useful work, your program might ask the user to resize the window. The border of your top-level window is considered fair game for window managers.

To set an attribute of a window, set the appropriate member of the *XSetWindowAttributes* structure and OR in the corresponding value bitmask in your subsequent calls to *XCreateWindow* and *XChangeWindowAttributes*, or use one of the other convenience functions that set the appropriate attribute. The symbols for the value mask bits and the *XSetWindowAttributes* structure are:

X/Open CAE Specification
                                                                   Stamp:XXXXXXXXXXXXXXXXXXXXXXXXX

```
/* Window attribute value mask bits */

    #define    CWBackPixmap        (1L<<0)
    #define    CWBackPixel         (1L<<1)
    #define    CWBorderPixmap      (1L<<2)
    #define    CWBorderPixel       (1L<<3)
    #define    CWBitGravity        (1L<<4)
    #define    CWWinGravity        (1L<<5)
    #define    CWBackingStore      (1L<<6)
    #define    CWBackingPlanes     (1L<<7)
    #define    CWBackingPixel      (1L<<8)
    #define    CWOverrideRedirect  (1L<<9)
    #define    CWSaveUnder         (1L<<10)
    #define    CWEventMask         (1L<<11)
    #define    CWDontPropagate     (1L<<12)
    #define    CWColormap          (1L<<13)
    #define    CWCursor            (1L<<14)

/* Values */

typedef struct {
    Pixmap background_pixmap;        /* background, None or
                                        ParentRelative */
    unsigned long background_pixel;  /* background pixel */
    Pixmap border_pixmap;            /* border of the window or
                                        CopyFromParent */
    unsigned long border_pixel;      /* border pixel value */
    int bit_gravity;                 /* one of bit gravity values */
    int win_gravity;                 /* one of the window gravity
                                        values */
    int backing_store;               /* NotUseful, WhenMapped, Always */
    unsigned long backing_planes;    /* planes to be preserved if
                                        possible */
    unsigned long backing_pixel;     /* value to use in restoring
                                        planes */
    Bool save_under;                 /* should bits under be saved?
                                        (popups) */
    long event_mask;                 /* set of events that should
                                        be saved */
    long do_not_propagate_mask;      /* set of events that should
                                        not propagate */
    Bool override_redirect;          /* boolean value for
                                        override_redirect */
    Colormap colormap;               /* color map to be associated
                                        with window */
    Cursor cursor;                   /* cursor to be displayed
                                        (or None) */
} XSetWindowAttributes;
```

The following lists the defaults for each window attribute and indicates whether the attribute is applicable to *InputOutput* and *InputOnly* windows:

Attribute	Default	*InputOutput*	*InputOnly*
background-pixmap	None	Yes	No
background-pixel	Undefined	Yes	No
border-pixmap	CopyFromParent	Yes	No
border-pixel	Undefined	Yes	No
bit-gravity	ForgetGravity	Yes	No
win-gravity	NorthWestGravity	Yes	Yes
backing-store	NotUseful	Yes	No
backing-planes	All ones	Yes	No
backing-pixel	zero	Yes	No
save-under	False	Yes	No
event-mask	empty set	Yes	Yes
do-not-propagate-mask	empty set	Yes	Yes
override-redirect	False	Yes	Yes
colormap	CopyFromParent	Yes	No
cursor	None	Yes	Yes

4.2.1   Background Attribute

Only *InputOutput* windows can have a background. You can set the background of an *InputOutput* window by using a pixel or a pixmap.

The background-pixmap attribute of a window specifies the pixmap to be used for a window's background. This pixmap can be of any size, although some sizes may be faster than others. The background-pixel attribute of a window specifies a pixel value used to paint a window's background in a single color.

You can set the background-pixmap to a pixmap, *None* (default), or *ParentRelative*. You can set the background-pixel of a window to any pixel value (no default). If you specify a background-pixel, it overrides either the default background-pixmap or any value you may have set in the background-pixmap. A pixmap of an undefined size that is filled with the background-pixel is used for the background. Range checking is not performed on the background pixel; it simply is truncated to the appropriate number of bits.

If you set the background-pixmap, it overrides the default. The background-pixmap and the window must have the same depth, or a *BadMatch* error results. If you set background-pixmap to *None*, the window has no defined background. If you set the background-pixmap to *ParentRelative*:

- The parent window's background-pixmap is used. The child window, however, must have the same depth as its parent, or a *BadMatch* error results.

- If the parent window has a background-pixmap of *None*, the window also has a background-pixmap of *None*.

- A copy of the parent window's background-pixmap is not made. The parent's background-pixmap is examined each time the child window's background-pixmap is required.

- The background tile origin always aligns with the parent window's background tile origin. If the background-pixmap is not *ParentRelative*, the background tile origin is the child window's origin.

Setting a new background, whether by setting background-pixmap or background-pixel, overrides any previous background. The background-pixmap can be freed immediately if no further explicit reference is made to it (the X server will keep a copy to use when needed). If you later draw into the pixmap used for the background, what happens is undefined because the X

implementation is free to make a copy of the pixmap or to use the same pixmap.

When no valid contents are available for regions of a window and either the regions are visible or the server is maintaining backing store, the server automatically tiles the regions with the window's background unless the window has a background of *None*. If the background is *None*, the previous screen contents from other windows of the same depth as the window are simply left in place as long as the contents come from the parent of the window or an inferior of the parent. Otherwise, the initial contents of the exposed regions are undefined. *Expose* events are then generated for the regions, even if the background-pixmap is *None* (see Section 11.9).

4.2.2   Border Attribute

Only *InputOutput* windows can have a border. You can set the border of an *InputOutput* window by using a pixel or a pixmap.

The border-pixmap attribute of a window specifies the pixmap to be used for a window's border. The border-pixel attribute of a window specifies a pixmap of undefined size filled with that pixel be used for a window's border. Range checking is not performed on the background pixel; it simply is truncated to the appropriate number of bits. The border tile origin is always the same as the background tile origin.

You can also set the border-pixmap to a pixmap of any size (some may be faster than others) or to *CopyFromParent* (default). You can set the border-pixel to any pixel value (no default).

If you set a border-pixmap, it overrides the default. The border-pixmap and the window must have the same depth, or a *BadMatch* error results. If you set the border-pixmap to *CopyFromParent*, the parent window's border-pixmap is copied. Subsequent changes to the parent window's border attribute do not affect the child window. However, the child window must have the same depth as the parent window, or a *BadMatch* error results.

The border-pixmap can be freed immediately if no further explicit reference is made to it. If you later draw into the pixmap used for the border, what happens is undefined because the X implementation is free either to make a copy of the pixmap or to use the same pixmap. If you specify a border-pixel, it overrides either the default border-pixmap or any value you may have set in the border-pixmap. All pixels in the window's border will be set to the border-pixel. Setting a new border, whether by setting border-pixel or by setting border-pixmap, overrides any previous border.

Output to a window is always clipped to the inside of the window. Therefore, graphics operations never affect the window border.

4.2.3   Gravity Attributes

The bit gravity of a window defines which region of the window should be retained when an *InputOutput* window is resized. The default value for the bit-gravity attribute is *ForgetGravity*. The window gravity of a window allows you to define how the *InputOutput* or *InputOnly* window should be repositioned if its parent is resized. The default value for the win-gravity attribute is *NorthWestGravity*.

If the inside width or height of a window is not changed and if the window is moved or its border is changed, then the contents of the window are not lost but move with the window. Changing the inside width or height of the window causes its contents to be moved or lost (depending on the bit-gravity of the window) and causes children to be reconfigured (depending on their win-gravity). For a change of width and height, the (x, y) pairs are defined:

Gravity Direction	Coordinates
NorthWestGravity	(0, 0)
NorthGravity	(Width/2, 0)
NorthEastGravity	(Width, 0)
WestGravity	(0, Height/2)
CenterGravity	(Width/2, Height/2)
EastGravity	(Width, Height/2)
SouthWestGravity	(0, Height)
SouthGravity	(Width/2, Height)
SouthEastGravity	(Width, Height)

When a window with one of these bit-gravity values is resized, the corresponding pair defines the change in position of each pixel in the window. When a window with one of these win-gravities has its parent window resized, the corresponding pair defines the change in position of the window within the parent. When a window is so repositioned, a *GravityNotify* event is generated (see Section 11.10.5).

A bit-gravity of *StaticGravity* indicates that the contents or origin should not move relative to the origin of the root window. If the change in size of the window is coupled with a change in position (x, y), then for bit-gravity the change in position of each pixel is (–x, –y), and for win-gravity the change in position of a child when its parent is so resized is (–x, –y). Note that *StaticGravity* still only takes effect when the width or height of the window is changed, not when the window is moved.

A bit-gravity of *ForgetGravity* indicates that the window's contents are always discarded after a size change, even if a backing store or save under has been requested. The window is tiled with its background and zero or more *Expose* events are generated. If no background is defined, the existing screen contents are not altered. Some X servers may also ignore the specified bit-gravity and always generate *Expose* events.

The contents and borders of inferiors are not affected by their parent's bit-gravity. A server is permitted to ignore the specified bit-gravity and use *Forget* instead.

A win-gravity of *UnmapGravity* is like *NorthWestGravity* (the window is not moved), except the child is also unmapped when the parent is resized, and an *UnmapNotify* event is generated.

4.2.4   Backing Store Attribute

Some implementations of the X server may choose to maintain the contents of *InputOutput* windows. If the X server maintains the contents of a window, the off-screen saved pixels are known as backing store. The backing store advises the X server on what to do with the contents of a window. The backing-store attribute can be set to *NotUseful* (default), *WhenMapped*, or *Always*.

A backing-store attribute of *NotUseful* advises the X server that maintaining contents is unnecessary, although some X implementations may still choose to maintain contents and, therefore, not generate *Expose* events. A backing-store attribute of *WhenMapped* advises the X server that maintaining contents of obscured regions when the window is mapped would be beneficial. In this case, the server may generate an *Expose* event when the window is created. A backing-store attribute of *Always* advises the X server that maintaining contents even when the window is unmapped would be beneficial. Even if the window is larger than its parent, this is a request to the X server to maintain complete contents, not just the region within the parent window boundaries. While the X server maintains the window's contents, *Expose* events normally are not generated, but the X server may stop maintaining contents at any time.

When the contents of obscured regions of a window are being maintained, regions obscured by noninferior windows are included in the destination of graphics requests (and source, when the window is the source). However, regions obscured by inferior windows are not included.

4.2.5   Save Under Flag

Some server implementations may preserve contents of *InputOutput* windows under other *InputOutput* windows. This is not the same as preserving the contents of a window for you. You may get better visual appeal if transient windows (for example, pop-up menus) request that the system preserve the screen contents under them, so the temporarily obscured applications do not have to repaint.

You can set the save-under flag to *True* or *False* (default). If save-under is *True*, the X server is advised that, when this window is mapped, saving the contents of windows it obscures would be beneficial.

4.2.6   Backing Planes and Backing Pixel Attributes

You can set backing planes to indicate (with bits set to 1) which bit planes of an *InputOutput* window hold dynamic data that must be preserved in backing store and during save unders. The default value for the backing-planes attribute is all bits set to 1. You can set backing pixel to specify what bits to use in planes not covered by backing planes. The default value for the backing-pixel attribute is all bits set to 0. The X server is free to save only the specified bit planes in the backing store or the save under and is free to regenerate the remaining planes with the specified pixel value. Any extraneous bits in these values (that is, those bits beyond the specified depth of the window) may be simply ignored. If you request backing store or save unders, you should use these members to minimize the amount of off-screen memory required to store your window.

4.2.7   Event Mask and Do Not Propagate Mask Attributes

The event mask defines which events the client is interested in for this *InputOutput* or *InputOnly* window (or, for some event types, inferiors of this window). The event mask is the bitwise inclusive OR of zero or more of the valid event mask bits. You can specify that no maskable events are reported by setting *NoEventMask* (default).

The do-not-propagate-mask attribute defines which events should not be propagated to ancestor windows when no client has the event type selected in this *InputOutput* or *InputOnly* window. The do-not-propagate-mask is the bitwise inclusive OR of zero or more of the following masks: *KeyPress*, *KeyRelease*, *ButtonPress*, *ButtonRelease*, *PointerMotion*, *Button1Motion*, *Button2Motion*, *Button3Motion*, *Button4Motion*, *Button5Motion* and *ButtonMotion*. You can specify that all events are propagated by setting *NoEventMask* (default).

4.2.8   Override Redirect Flag

To control window placement or to add decoration, a window manager often needs to intercept (redirect) any map or configure request. Pop-up windows, however, often need to be mapped without a window manager getting in the way. To control whether an *InputOutput* or *InputOnly* window is to ignore these structure control facilities, use the override-redirect flag.

The override-redirect flag specifies whether map and configure requests on this window should override a *SubstructureRedirectMask* on the parent. You can set the override-redirect flag to *True* or *False* (default). Window managers use this information to avoid tampering with pop-up windows (see also Chapter 15).

4.2.9   Colormap Attribute

The colormap attribute specifies which colormap best reflects the true colors of the *InputOutput* window. The colormap must have the same visual type as the window, or a *BadMatch* error results. X servers capable of supporting multiple hardware colormaps can use this information, and window managers can use it for calls to *XInstallColormap*. You can set the colormap attribute to a colormap or to *CopyFromParent* (default).

If you set the colormap to *CopyFromParent*, the parent window's colormap is copied and used by its child. However, the child window must have the same visual type as the parent, or a *BadMatch* error results. The parent window must not have a colormap of *None*, or a *BadMatch* error results. The colormap is copied by sharing the colormap object between the child and parent, not by making a complete copy of the colormap contents. Subsequent changes to the parent window's colormap attribute do not affect the child window.

4.2.10   Cursor Attribute

The cursor attribute specifies which cursor is to be used when the pointer is in the *InputOutput* or *InputOnly* window. You can set the cursor to a cursor or *None* (default).

If you set the cursor to *None*, the parent's cursor is used when the pointer is in the *InputOutput* or *InputOnly* window, and any change in the parent's cursor will cause an immediate change in the displayed cursor. By calling *XFreeCursor*, the cursor can be freed immediately as long as no further explicit reference to it is made.

4.3   Creating Windows

Xlib provides basic ways for creating windows, and toolkits often supply higher-level functions specifically for creating and placing top-level windows, which are discussed in the appropriate toolkit documentation. If you do not use a toolkit, however, you must provide some standard information or hints for the window manager by using the Xlib inter-client communication functions (see Chapter 15).

If you use Xlib to create your own top-level windows (direct children of the root window), you must observe the following rules so that all applications interact reasonably across the different styles of window management:

- You must never fight with the window manager for the size or placement of your top-level window.

- You must be able to deal with whatever size window you get, even if this means that your application just prints a message like ''Please make me bigger'' in its window.

- You should only attempt to resize or move top-level windows in direct response to a user request. If a request to change the size of a top-level window fails, you must be prepared to live with what you get. You are free to resize or move the children of top-level windows as necessary. (Toolkits often have facilities for automatic relayout.)

- If you do not use a toolkit that automatically sets standard window properties, you should set these properties for top-level windows before mapping them.

For further information, see Chapter 15 and the **ICCCM** specification.

XCreateWindow is the more general function that allows you to set specific window attributes when you create a window. *XCreateSimpleWindow* creates a window that inherits its attributes from its parent window.

The X server acts as if *InputOnly* windows do not exist for the purposes of graphics requests, exposure processing and *VisibilityNotify* events. An *InputOnly* window cannot be used as a drawable (that is, as a source or destination for graphics requests). *InputOnly* and *InputOutput* windows act identically in other respects (properties, grabs, input control, and so on). Extension packages can define other classes of windows.

To create an unmapped window and set its window attributes, use *XCreateWindow*.

```
Window XCreateWindow(display, parent, x, y,
        width, height, border_width, depth,
        class, visual, valuemask, attributes)
    Display *display;
    Window parent;
    int x, y;
    unsigned int width, height;
    unsigned int border_width;
    int depth;
    unsigned int class;
    Visual *visual
    unsigned long valuemask;
    XSetWindowAttributes *attributes;
```

display   Specifies the connection to the X server.

parent   Specifies the parent window.

x
y   Specify the x and y coordinates, which are the top-left outside corner of the
                             created window's borders and are relative to the inside of the parent window's
                             borders.

width
height   Specify the width and height, which are the created window's inside
                             dimensions and do not include the created window's borders. The dimensions
                             must be non-zero, or a *BadValue* error results.

border_width   Specifies the width of the created window's border in pixels.

depth   Specifies the window's depth. A depth of *CopyFromParent* means the depth is
                             taken from the parent.

class   Specifies the created window's class. You can pass *InputOutput*, *InputOnly* or
                             CopyFromParent. A class of *CopyFromParent* means the class is taken from the
                             parent.

visual   Specifies the visual type. A visual of *CopyFromParent* means the visual type is
                             taken from the parent.

valuemask   Specifies which window attributes are defined in the attributes argument.
                             This mask is the bitwise inclusive OR of the valid attribute mask bits. If
                             valuemask is zero, the attributes are ignored and are not referenced.

attributes   Specifies the structure from which the values (as specified by the value mask)
                             are to be taken. The value mask should have the appropriate bits set to
                             indicate which attributes have been set in the structure.

The *XCreateWindow* function creates an unmapped subwindow for a specified parent window,
returns the window ID of the created window, and causes the X server to generate a *CreateNotify*
event. The created window is placed on top in the stacking order with respect to siblings.

The coordinate system has the X axis horizontal and the Y axis vertical, with the origin [0, 0] at
the upper left. Coordinates are integral, in terms of pixels, and coincide with pixel centers. Each
window and pixmap has its own coordinate system. For a window, the origin is inside the
border at the inside upper left.

The border_width for an *InputOnly* window must be zero, or a *BadMatch* error results. For class
InputOutput, the visual type and depth must be a combination supported for the screen, or a
BadMatch error results. The depth need not be the same as the parent, but the parent must not be
a window of class *InputOnly*, or a *BadMatch* error results. For an *InputOnly* window, the depth
must be zero, and the visual must be one supported by the screen. If either condition is not met,
a *BadMatch* error results. The parent window, however, may have any depth and class. If you
specify any invalid window attribute for a window, a *BadMatch* error results.

The created window is not yet displayed (mapped) on the user's display. To display the
window, call *XMapWindow*. The new window initially uses the same cursor as its parent. A
new cursor can be defined for the new window by calling *XDefineCursor*. The window will not
be visible on the screen unless it and all of its ancestors are mapped and it is not obscured by any
of its ancestors.

XCreateWindow can generate *BadAlloc*, *BadColor*, *BadCursor*, *BadMatch*, *BadPixmap*, *BadValue* and
BadWindow errors.

To create an unmapped *InputOutput* subwindow of a given parent window, use
XCreateSimpleWindow.

```
Window XCreateSimpleWindow(display, parent, x,
        y, width, height, border_width,
        border, background)
    Display *display;
    Window parent;
    int x, y;
    unsigned int width, height;
    unsigned int border_width;
    unsigned long border;
    unsigned long background;
```

display   Specifies the connection to the X server.

parent   Specifies the parent window.

x
y   Specify the x and y coordinates, which are the top-left outside corner of the
                    new window's borders and are relative to the inside of the parent window's
                    borders.

width
height   Specify the width and height, which are the created window's inside
                    dimensions and do not include the created window's borders. The dimensions
                    must be non-zero, or a *BadValue* error results.

border_width   Specifies the width of the created window's border in pixels.

border   Specifies the border pixel value of the window.

background   Specifies the background pixel value of the window.

The *XCreateSimpleWindow* function creates an unmapped *InputOutput* subwindow for a specified
parent window, returns the window ID of the created window, and causes the X server to
generate a *CreateNotify* event. The created window is placed on top in the stacking order with
respect to siblings. Any part of the window that extends outside its parent window is clipped.
The border_width for an *InputOnly* window must be zero, or a *BadMatch* error results.
XCreateSimpleWindow inherits its depth, class and visual from its parent. All other window
attributes, except background and border, have their default values.

XCreateSimpleWindow can generate *BadAlloc*, *BadMatch*, *BadValue* and *BadWindow* errors.

4.4   Destroying Windows

Xlib provides functions that you can use to destroy a window or destroy all subwindows of a window.

To destroy a window and all of its subwindows, use *XDestroyWindow*.

```
XDestroyWindow(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XDestroyWindow* function destroys the specified window as well as all of its subwindows and causes the X server to generate a *DestroyNotify* event for each window. The window should never be referenced again. If the window specified by the w argument is mapped, it is unmapped automatically. The ordering of the *DestroyNotify* events is such that for any given window being destroyed, *DestroyNotify* is generated on any inferiors of the window before being generated on the window itself. The ordering among siblings and across subhierarchies is not otherwise constrained. If the window you specified is a root window, no windows are destroyed. Destroying a mapped window will generate *Expose* events on other windows that were obscured by the window being destroyed.

XDestroyWindow can generate a *BadWindow* error.

To destroy all subwindows of a specified window, use *XDestroySubwindows*.

```
XDestroySubwindows(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XDestroySubwindows* function destroys all inferior windows of the specified window, in bottom-to-top stacking order. It causes the X server to generate a *DestroyNotify* event for each window. If any mapped subwindows were actually destroyed, *XDestroySubwindows* causes the X server to generate *Expose* events on the specified window. This is much more efficient than deleting many windows one at a time because much of the work need be performed only once for all of the windows, rather than for each window. The subwindows should never be referenced again.

XDestroySubwindows can generate a *BadWindow* error.

4.5   Mapping Windows

A window is considered mapped if an *XMapWindow* call has been made on it. It may not be visible on the screen for one of the following reasons:

- It is obscured by another opaque window.

- One of its ancestors is not mapped.

- It is entirely clipped by an ancestor.

Expose events are generated for the window when part or all of it becomes visible on the screen. A client receives the *Expose* events only if it has asked for them. Windows retain their position in the stacking order when they are unmapped.

A window manager may want to control the placement of subwindows. If *SubstructureRedirectMask* has been selected by a window manager on a parent window (usually a root window), a map request initiated by other clients on a child window is not performed, and the window manager is sent a *MapRequest* event. However, if the override-redirect flag on the child had been set to *True* (usually only on pop-up menus), the map request is performed.

A tiling window manager might decide to reposition and resize other clients' windows and then decide to map the window to its final location. A window manager that wants to provide decoration might reparent the child into a frame first. For further information, see Section 4.2.8 and Section 11.10. Only a single client at a time can select for *SubstructureRedirectMask*.

Similarly, a single client can select for *ResizeRedirectMask* on a parent window. Then, any attempt to resize the window by another client is suppressed, and the client receives a *ResizeRequest* event.

To map a given window, use *XMapWindow*.

```
XMapWindow(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XMapWindow* function maps the window and all of its subwindows that have had map requests. Mapping a window that has an unmapped ancestor does not display the window but marks it as eligible for display when the ancestor becomes mapped. Such a window is called unviewable. When all its ancestors are mapped, the window becomes viewable and will be visible on the screen if it is not obscured by another window. This function has no effect if the window is already mapped.

If the override-redirect of the window is *False* and if some other client has selected *SubstructureRedirectMask* on the parent window, then the X server generates a *MapRequest* event, and the *XMapWindow* function does not map the window. Otherwise, the window is mapped, and the X server generates a *MapNotify* event.

If the window becomes viewable and no earlier contents for it are remembered, the X server tiles the window with its background. If the window's background is undefined, the existing screen contents are not altered, and the X server generates zero or more *Expose* events. If backing-store was maintained while the window was unmapped, no *Expose* events are generated. If backing-store will now be maintained, a full-window exposure is always generated. Otherwise, only visible regions may be reported. Similar tiling and exposure take place for any newly viewable inferiors.

If the window is an *InputOutput* window, *XMapWindow* generates *Expose* events on each *InputOutput* window that it causes to be displayed. If the client maps and paints the window and if the client begins processing events, the window is painted twice. To avoid this, first ask for *Expose* events and then map the window, so the client processes input events as usual. The event list will include *Expose* for each window that has appeared on the screen. The client's normal response to an *Expose* event should be to repaint the window. This method usually leads to simpler programs and to proper interaction with window managers.

XMapWindow can generate a *BadWindow* error.

To map and raise a window, use *XMapRaised*.

```
XMapRaised(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XMapRaised* function essentially is similar to *XMapWindow* in that it maps the window and all of its subwindows that have had map requests. However, it also raises the specified window to the top of the stack. For additional information, see *XMapWindow*.

XMapRaised can generate multiple *BadWindow* errors.

To map all subwindows for a specified window, use *XMapSubwindows*.

```
XMapSubwindows(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XMapSubwindows* function maps all subwindows for a specified window in top-to-bottom stacking order. The X server generates *Expose* events on each newly displayed window. This may be much more efficient than mapping many windows one at a time because the server needs to perform much of the work only once, for all of the windows, rather than for each window.

XMapSubwindows can generate a *BadWindow* error.

4.6   Unmapping Windows

Xlib provides functions that you can use to unmap a window or all subwindows.

To unmap a window, use *XUnmapWindow*.

```
XUnmapWindow(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XUnmapWindow* function unmaps the specified window and causes the X server to generate an *UnmapNotify* event. If the specified window is already unmapped, *XUnmapWindow* has no effect. Normal exposure processing on formerly obscured windows is performed. Any child window will no longer be visible until another map call is made on the parent. In other words, the subwindows are still mapped but are not visible until the parent is mapped. Unmapping a window will generate *Expose* events on windows that were formerly obscured by it.

XUnmapWindow can generate a *BadWindow* error.

To unmap all subwindows for a specified window, use *XUnmapSubwindows*.

```
XUnmapSubwindows(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XUnmapSubwindows* function unmaps all subwindows for the specified window in bottom-to-top stacking order. It causes the X server to generate an *UnmapNotify* event on each subwindow and *Expose* events on formerly obscured windows. Using this function is much more efficient than unmapping multiple windows one at a time because the server needs to perform much of the work only once, for all of the windows, rather than for each window.

XUnmapSubwindows can generate a *BadWindow* error.

4.7   Configuring Windows

Xlib provides functions that you can use to move a window, resize a window, move and resize a window, or change a window's border width. To change one of these parameters, set the appropriate member of the *XWindowChanges* structure and OR in the corresponding value mask in subsequent calls to *XConfigureWindow*. The symbols for the value mask bits and the *XWindowChanges* structure are:

```
/* Configure window value mask bits */

    #define   CWX              (1<<0)
    #define   CWY              (1<<1)
    #define   CWWidth          (1<<2)
    #define   CWHeight         (1<<3)
    #define   CWBorderWidth    (1<<4)
    #define   CWSibling        (1<<5)
    #define   CWStackMode      (1<<6)


/* Values */

typedef struct {
    int x, y;
    int width, height;
    int border_width;
    Window sibling;
    int stack_mode;
} XWindowChanges;
```

The x and y members are used to set the window's x and y coordinates, which are relative to the parent's origin and indicate the position of the upper-left outer corner of the window. The width and height members are used to set the inside size of the window, not including the border, and must be non-zero, or a *BadValue* error results. Attempts to configure a root window have no effect.

The border_width member is used to set the width of the border in pixels. Note that setting just the border width leaves the outer-left corner of the window in a fixed position but moves the absolute position of the window's origin. If you attempt to set the border-width attribute of an *InputOnly* window non-zero, a *BadMatch* error results.

The sibling member is used to set the sibling window for stacking operations. The stack_mode member is used to set how the window is to be restacked and can be set to *Above*, *Below*, *TopIf*, *BottomIf* or *Opposite*.

If the override-redirect flag of the window is *False* and if some other client has selected *SubstructureRedirectMask* on the parent, the X server generates a *ConfigureRequest* event, and no further processing is performed. Otherwise, if some other client has selected *ResizeRedirectMask* on the window and the inside width or height of the window is being changed, a *ResizeRequest* event is generated, and the current inside width and height are used instead. Note that the override-redirect flag of the window has no effect on *ResizeRedirectMask* and that *SubstructureRedirectMask* on the parent has precedence over *ResizeRedirectMask* on the window.

When the geometry of the window is changed as specified, the window is restacked among siblings, and a *ConfigureNotify* event is generated if the state of the window actually changes. *GravityNotify* events are generated after *ConfigureNotify* events. If the inside width or height of the window has actually changed, children of the window are affected as specified.

If a window's size actually changes, the window's subwindows move according to their window gravity. Depending on the window's bit gravity, the contents of the window also may be moved (see Section 4.2.3).

If regions of the window were obscured but now are not, exposure processing is performed on these formerly obscured windows, including the window itself and its inferiors. As a result of increasing the width or height, exposure processing is also performed on any new regions of the window and any regions where window contents are lost.

The restack check (specifically, the computation for *BottomIf*, *TopIf* and *Opposite*) is performed with respect to the window's final size and position (as controlled by the other arguments of the request), not its initial position. If a sibling is specified without a stack_mode, a *BadMatch* error results.

If a sibling and a stack_mode are specified, the window is restacked as follows:

Above   The window is placed just above the sibling.

Below   The window is placed just below the sibling.

TopIf   If the sibling occludes the window, the window is placed at the top of the stack.

BottomIf   If the window occludes the sibling, the window is placed at the bottom of the stack.

Opposite   If the sibling occludes the window, the window is placed at the top of the stack. If the window occludes the sibling, the window is placed at the bottom of the stack.

If a stack_mode is specified but no sibling is specified, the window is restacked as follows:

Above   The window is placed at the top of the stack.

Below   The window is placed at the bottom of the stack.

TopIf   If any sibling occludes the window, the window is placed at the top of the stack.

BottomIf   If the window occludes any sibling, the window is placed at the bottom of the stack.

Opposite   If any sibling occludes the window, the window is placed at the top of the stack. If the window occludes any sibling, the window is placed at the bottom of the stack.

Attempts to configure a root window have no effect.

To configure a window's size, location, stacking or border, use *XConfigureWindow*.

```
XConfigureWindow(display, w, value_mask, values)
      Display *display;
      Window w;
      unsigned int value_mask;
      XWindowChanges *values;
```

display   Specifies the connection to the X server.

w   Specifies the window to be reconfigured.

value_mask   Specifies which values are to be set using information in the values structure. This mask is the bitwise inclusive OR of the valid configure window values

bits.

values   Specifies the *XWindowChanges* structure.

The *XConfigureWindow* function uses the values specified in the *XWindowChanges* structure to reconfigure a window's size, position, border and stacking order. Values not specified are taken from the existing geometry of the window.

If a sibling is specified without a stack_mode or if the window is not actually a sibling, a *BadMatch* error results. Note that the computations for *BottomIf*, *TopIf* and *Opposite* are performed with respect to the window's final geometry (as controlled by the other arguments passed to *XConfigureWindow*), not its initial geometry. Any backing store contents of the window, its inferiors, and other newly visible windows are either discarded or changed to reflect the current screen contents (depending on the implementation).

XConfigureWindow can generate *BadMatch*, *BadValue* and *BadWindow* errors.

To move a window without changing its size, use *XMoveWindow*.

```
XMoveWindow(display, w, x, y)
      Display *display;
      Window w;
      int x, y;
```

display   Specifies the connection to the X server.

w   Specifies the window to be moved.

x

y   Specify the x and y coordinates, which define the new location of the top-left pixel of the window's border or the window itself if it has no border.

The *XMoveWindow* function moves the specified window to the specified x and y coordinates, but it does not change the window's size, raise the window, or change the mapping state of the window. Moving a mapped window may or may not lose the window's contents depending on if the window is obscured by nonchildren and if no backing store exists. If the contents of the window are lost, the X server generates *Expose* events. Moving a mapped window generates *Expose* events on any formerly obscured windows.

If the override-redirect flag of the window is *False* and some other client has selected *SubstructureRedirectMask* on the parent, the X server generates a *ConfigureRequest* event, and no further processing is performed. Otherwise, the window is moved.

XMoveWindow can generate a *BadWindow* error.

To change a window's size without changing the upper-left coordinate, use *XResizeWindow*.

```
XResizeWindow(display, w, width, height)
      Display *display;
      Window w;
      unsigned int width, height;
```

display   Specifies the connection to the X server.

w   Specifies the window.

width

height   Specify the width and height, which are the interior dimensions of the window after the call completes.

The *XResizeWindow* function changes the inside dimensions of the specified window, not including its borders. This function does not change the window's upper-left coordinate or the origin and does not restack the window. Changing the size of a mapped window may lose its contents and generate *Expose* events. If a mapped window is made smaller, changing its size generates *Expose* events on windows that the mapped window formerly obscured.

If the override-redirect flag of the window is *False* and some other client has selected *SubstructureRedirectMask* on the parent, the X server generates a *ConfigureRequest* event, and no further processing is performed. If either width or height is zero, a *BadValue* error results.

XResizeWindow can generate *BadValue* and *BadWindow* errors.

To change the size and location of a window, use *XMoveResizeWindow*.

```
XMoveResizeWindow(display, w, x, y,
        width, height)
    Display *display;
    Window w;
    int x, y;
    unsigned int width, height;
```

display   Specifies the connection to the X server.

w   Specifies the window to be reconfigured.

x
y   Specify the x and y coordinates, which define the new position of the window relative to its parent.

width
height   Specify the width and height, which define the interior size of the window.

The *XMoveResizeWindow* function changes the size and location of the specified window without raising it. Moving and resizing a mapped window may generate an *Expose* event on the window. Depending on the new size and location parameters, moving and resizing a window may generate *Expose* events on windows that the window formerly obscured.

If the override-redirect flag of the window is *False* and some other client has selected *SubstructureRedirectMask* on the parent, the X server generates a *ConfigureRequest* event, and no further processing is performed. Otherwise, the window size and location are changed.

XMoveResizeWindow can generate *BadValue* and *BadWindow* errors.

To change the border width of a given window, use *XSetWindowBorderWidth*.

```
XSetWindowBorderWidth(display, w, width)
    Display *display;
    Window w;
    unsigned int width;
```

display   Specifies the connection to the X server.

w   Specifies the window.

width   Specifies the width of the window border.

The *XSetWindowBorderWidth* function sets the specified window's border width to the specified width.

XSetWindowBorderWidth can generate a *BadWindow* error.

4.8   Changing Window Stacking Order

Xlib provides functions that you can use to raise, lower, circulate or restack windows.

To raise a window so that no sibling window obscures it, use *XRaiseWindow*.

```
XRaiseWindow(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XRaiseWindow* function raises the specified window to the top of the stack so that no sibling window obscures it. If the windows are regarded as overlapping sheets of paper stacked on a desk, then raising a window is analogous to moving the sheet to the top of the stack but leaving its x and y location on the desk constant. Raising a mapped window may generate *Expose* events for the window and any mapped subwindows that were formerly obscured.

If the override-redirect attribute of the window is *False* and some other client has selected *SubstructureRedirectMask* on the parent, the X server generates a *ConfigureRequest* event, and no processing is performed. Otherwise, the window is raised.

XRaiseWindow can generate a *BadWindow* error.

To lower a window so that it does not obscure any sibling windows, use *XLowerWindow*.

```
XLowerWindow(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XLowerWindow* function lowers the specified window to the bottom of the stack so that it does not obscure any sibling windows. If the windows are regarded as overlapping sheets of paper stacked on a desk, then lowering a window is analogous to moving the sheet to the bottom of the stack but leaving its x and y location on the desk constant. Lowering a mapped window will generate *Expose* events on any windows it formerly obscured.

If the override-redirect attribute of the window is *False* and some other client has selected *SubstructureRedirectMask* on the parent, the X server generates a *ConfigureRequest* event, and no processing is performed. Otherwise, the window is lowered to the bottom of the stack.

XLowerWindow can generate a *BadWindow* error.

To circulate a subwindow up or down, use *XCirculateSubwindows*.

```
XCirculateSubwindows(display, w, direction)
      Display *display;
      Window w;
      int direction;
```

display   Specifies the connection to the X server.

w   Specifies the window.

direction   Specifies the direction (up or down) that you want to circulate the window.
                     You can pass *RaiseLowest* or *LowerHighest*.

The *XCirculateSubwindows* function circulates children of the specified window in the specified direction. If you specify *RaiseLowest*, *XCirculateSubwindows* raises the lowest mapped child (if any) that is occluded by another child to the top of the stack. If you specify *LowerHighest*, *XCirculateSubwindows* lowers the highest mapped child (if any) that occludes another child to the bottom of the stack. Exposure processing is then performed on formerly obscured windows. If some other client has selected *SubstructureRedirectMask* on the window, the X server generates a *CirculateRequest* event, and no further processing is performed. If a child is actually restacked, the X server generates a *CirculateNotify* event.

XCirculateSubwindows can generate *BadValue* and *BadWindow* errors.

To raise the lowest mapped child of a window that is partially or completely occluded by another child, use *XCirculateSubwindowsUp*.

```
XCirculateSubwindowsUp(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XCirculateSubwindowsUp* function raises the lowest mapped child of the specified window that is partially or completely occluded by another child. Completely unobscured children are not affected. This is a convenience function equivalent to *XCirculateSubwindows* with *RaiseLowest* specified.

XCirculateSubwindowsUp can generate a *BadWindow* error.

To lower the highest mapped child of a window that partially or completely occludes another child, use *XCirculateSubwindowsDown*.

```
XCirculateSubwindowsDown(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XCirculateSubwindowsDown* function lowers the highest mapped child of the specified window that partially or completely occludes another child. Completely unobscured children are not affected. This is a convenience function equivalent to *XCirculateSubwindows* with *LowerHighest* specified.

XCirculateSubwindowsDown can generate a *BadWindow* error.

To restack a set of windows from top to bottom, use *XRestackWindows*.

```
XRestackWindows(display, windows, nwindows);
      Display *display;
      Window windows[];
      int nwindows;
```

display   Specifies the connection to the X server.

windows   Specifies an array containing the windows to be restacked.

nwindows   Specifies the number of windows to be restacked.

The *XRestackWindows* function restacks the windows in the order specified, from top to bottom. The stacking order of the first window in the windows array is unaffected, but the other

windows in the array are stacked underneath the first window, in the order of the array. The stacking order of the other windows is not affected. For each window in the window array that is not a child of the specified window, a *BadMatch* error results.

If the override-redirect attribute of a window is *False* and some other client has selected *SubstructureRedirectMask* on the parent, the X server generates *ConfigureRequest* events for each window whose override-redirect flag is not set, and no further processing is performed. Otherwise, the windows will be restacked in top to bottom order.

XRestackWindows can generate a *BadWindow* error.

4.9   Changing Window Attributes

Xlib provides functions that you can use to set window attributes. *XChangeWindowAttributes* is
the more general function that allows you to set one or more window attributes provided by the
XSetWindowAttributes structure. The other functions described in this section allow you to set
one specific window attribute, such as a window's background.

To change one or more attributes for a given window, use *XChangeWindowAttributes*.

```
XChangeWindowAttributes(display, w, valuemask, attributes)
      Display *display;
      Window w;
      unsigned long valuemask;
      XSetWindowAttributes *attributes;
```

display   Specifies the connection to the X server.

w   Specifies the window.

valuemask   Specifies which window attributes are defined in the attributes argument.
                   This mask is the bitwise inclusive OR of the valid attribute mask bits. If
                   valuemask is zero, the attributes are ignored and are not referenced. The
                   values and restrictions are the same as for *XCreateWindow*.

attributes   Specifies the structure from which the values (as specified by the value mask)
                   are to be taken. The value mask should have the appropriate bits set to
                   indicate which attributes have been set in the structure (see Section 4.2).

Depending on the valuemask, the *XChangeWindowAttributes* function uses the window attributes
in the *XSetWindowAttributes* structure to change the specified window attributes. Changing the
background does not cause the window contents to be changed. To repaint the window and its
background, use *XClearWindow*. Setting the border or changing the background such that the
border tile origin changes causes the border to be repainted. Changing the background of a root
window to *None* or *ParentRelative* restores the default background pixmap. Changing the border
of a root window to *CopyFromParent* restores the default border pixmap. Changing the win-
gravity does not affect the current position of the window. Changing the backing-store of an
obscured window to *WhenMapped* or *Always*, or changing the backing-planes, backing-pixel or
save-under of a mapped window may have no immediate effect. Changing the colormap of a
window (that is, defining a new map, not changing the contents of the existing map) generates a
ColormapNotify event. Changing the colormap of a visible window may have no immediate
effect on the screen because the map may not be installed (see *XInstallColormap*). Changing the
cursor of a root window to *None* restores the default cursor. Whenever possible, you are
encouraged to share colormaps.

Multiple clients can select input on the same window. Their event masks are maintained
separately. When an event is generated, it is reported to all interested clients. However, only
one client at a time can select for *SubstructureRedirectMask*, *ResizeRedirectMask* and
ButtonPressMask. If a client attempts to select any of these event masks and some other client
has already selected one, a *BadAccess* error results. There is only one do-not-propagate-mask for
a window, not one per client.

XChangeWindowAttributes can generate *BadAccess*, *BadColor*, *BadCursor*, *BadMatch*, *BadPixmap*,
BadValue and *BadWindow* errors.

To set the background of a window to a given pixel, use *XSetWindowBackground*.

```
XSetWindowBackground(display, w, background_pixel)
      Display *display;
      Window w;
      unsigned long background_pixel;
```

display   Specifies the connection to the X server.

w   Specifies the window.

background_pixel   Specifies the pixel that is to be used for the background.

The *XSetWindowBackground* function sets the background of the window to the specified pixel value. Changing the background does not cause the window contents to be changed. *XSetWindowBackground* uses a pixmap of undefined size filled with the pixel value you passed. If you try to change the background of an *InputOnly* window, a *BadMatch* error results.

XSetWindowBackground can generate *BadMatch* and *BadWindow* errors.

To set the background of a window to a given pixmap, use *XSetWindowBackgroundPixmap*.

```
XSetWindowBackgroundPixmap(display, w, background_pixmap)
      Display *display;
      Window w;
      Pixmap background_pixmap;
```

display   Specifies the connection to the X server.

w   Specifies the window.

background_pixmap Specifies the background pixmap, *ParentRelative* or *None*.

The *XSetWindowBackgroundPixmap* function sets the background pixmap of the window to the specified pixmap. The background pixmap can immediately be freed if no further explicit references to it are to be made. If *ParentRelative* is specified, the background pixmap of the window's parent is used, or on the root window, the default background is restored. If you try to change the background of an *InputOnly* window, a *BadMatch* error results. If the background is set to *None*, the window has no defined background.

XSetWindowBackgroundPixmap can generate *BadMatch*, *BadPixmap* and *BadWindow* errors.

Note: *XSetWindowBackground* and *XSetWindowBackgroundPixmap* do not change the current contents of the window.

To change and repaint a window's border to a given pixel, use *XSetWindowBorder*.

```
XSetWindowBorder(display, w, border_pixel)
      Display *display;
      Window w;
      unsigned long border_pixel;
```

display   Specifies the connection to the X server.

w   Specifies the window.

border_pixel   Specifies the entry in the colormap.

The *XSetWindowBorder* function sets the border of the window to the pixel value you specify. If you attempt to perform this on an *InputOnly* window, a *BadMatch* error results.

XSetWindowBorder can generate *BadMatch* and *BadWindow* errors.

To change and repaint the border tile of a given window, use *XSetWindowBorderPixmap*.

```
XSetWindowBorderPixmap(display, w, border_pixmap)
      Display *display;
      Window w;
      Pixmap border_pixmap;
```

display   Specifies the connection to the X server.

w   Specifies the window.

border_pixmap   Specifies the border pixmap or *CopyFromParent*.

The *XSetWindowBorderPixmap* function sets the border pixmap of the window to the pixmap you specify. The border pixmap can be freed immediately if no further explicit references to it are to be made. If you specify *CopyFromParent*, a copy of the parent window's border pixmap is used. If you attempt to perform this on an *InputOnly* window, a *BadMatch* error results.

XSetWindowBorderPixmap can generate *BadMatch*, *BadPixmap* and *BadWindow* errors.

To set the colormap of a given window, use *XSetWindowColormap*.

```
XSetWindowColormap(display, w, colormap)
      Display *display;
      Window w;
      Colormap colormap;
```

display   Specifies the connection to the X server.

w   Specifies the window.

colormap   Specifies the colormap.

The *XSetWindowColormap* function sets the specified colormap of the specified window. The colormap must have the same visual type as the window, or a *BadMatch* error results.

XSetWindowColormap can generate *BadColor*, *BadMatch* and *BadWindow* errors.

To define which cursor will be used in a window, use *XDefineCursor*.

```
XDefineCursor(display, w, cursor)
      Display *display;
      Window w;
      Cursor cursor;
```

display   Specifies the connection to the X server.

w   Specifies the window.

cursor   Specifies the cursor that is to be displayed or *None*.

If a cursor is set, it will be used when the pointer is in the window. If the cursor is *None*, it is equivalent to *XUndefineCursor*.

XDefineCursor can generate *BadCursor* and *BadWindow* errors.

To undefine the cursor in a given window, use *XUndefineCursor*.

```
XUndefineCursor(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XUndefineCursor* function undoes the effect of a previous *XDefineCursor* for this window. When the pointer is in the window, the parent's cursor will now be used. On the root window, the default cursor is restored.

XUndefineCursor can generate a *BadWindow* error.

Window Information Functions

After you connect the display to the X server and create a window, you can use the Xlib window information functions to:

- obtain information about a window

- translate screen coordinates

- manipulate property lists

- obtain and change window properties

- manipulate selections.

5.1   Obtaining Window Information

Xlib provides functions that you can use to obtain information about the window tree, the window's current attributes, the window's current geometry, or the current pointer coordinates. Because they are most frequently used by window managers, these functions all return a status to indicate whether the window still exists.

To obtain the parent, a list of children, and number of children for a given window, use *XQueryTree*.

```
Status XQueryTree(display, w, root_return,
        parent_return, children_return, nchildren_return)
    Display *display;
    Window w;
    Window *root_return;
    Window *parent_return;
    Window **children_return;
    unsigned int *nchildren_return;
```

display   Specifies the connection to the X server.

w   Specifies the window whose list of children, root, parent, and number of children you want to obtain.

root_return   Returns the root window.

parent_return   Returns the parent window.

children_return   Returns the list of children.

nchildren_return   Returns the number of children.

The *XQueryTree* function returns the root ID, the parent window ID, a pointer to the list of children windows, and the number of children in the list for the specified window. The children are listed in current stacking order, from bottommost (first) to topmost (last). *XQueryTree* returns zero if it fails and non-zero if it succeeds. To free this list when it is no longer needed, use *XFree*.

XQueryTree can generate a *BadWindow* error.

To obtain the current attributes of a given window, use *XGetWindowAttributes*.

```
Status XGetWindowAttributes(display, w, window_attributes_return)
    Display *display;
    Window w;
    XWindowAttributes *window_attributes_return;
```

display   Specifies the connection to the X server.

w   Specifies the window whose current attributes you want to obtain.

window_attributes_return
                     Returns the specified window's attributes in the *XWindowAttributes* structure.

The *XGetWindowAttributes* function returns the current attributes for the specified window to an *XWindowAttributes* structure.

```
typedef struct {
    int x, y;                       /* location of window */
    int width, height;              /* width and height of window */
    int border_width;               /* border width of window */
    int depth;                      /* depth of window */
    Visual *visual;                 /* the associated visual structure */
    Window root;                    /* root of screen containing window */
    int class;                      /* InputOutput, InputOnly*/
    int bit_gravity;                /* one of the bit gravity values */
    int win_gravity;                /* one of the window gravity values */
    int backing_store;              /* NotUseful, WhenMapped, Always */
    unsigned long backing_planes;   /* planes to be preserved if
                                       possible */
    unsigned long backing_pixel;    /* value to be used when restoring
                                       planes */
    Bool save_under;                /* boolean, should bits under be
                                       saved? */
    Colormap colormap;              /* color map to be associated with
                                       window */
    Bool map_installed;             /* boolean, is color map currently
                                       installed */
    int map_state;                  /* IsUnmapped, IsUnviewable,
                                       IsViewable */
    long all_event_masks;           /* set of events all people have
                                       interest in*/
    long your_event_mask;           /* my event mask */
    long do_not_propagate_mask;     /* set of events that should not
                                       propagate */
    Bool override_redirect;         /* boolean value for
                                       override-redirect */
    Screen *screen;                 /* back pointer to correct screen */
} XWindowAttributes;
```

The x and y members are set to the upper-left outer corner relative to the parent window's origin. The width and height members are set to the inside size of the window, not including the border. The border_width member is set to the window's border width in pixels. The depth member is set to the depth of the window (that is, bits per pixel for the object). The visual member is a pointer to the screen's associated *Visual* structure. The root member is set to the root window of the screen containing the window. The class member is set to the window's class and can be either *InputOutput* or *InputOnly*.

The bit_gravity member is set to the window's bit gravity and can be one of the following:

<div align="center">

ForgetGravity   *EastGravity*
NorthWestGravity   *SouthWestGravity*
NorthGravity   *SouthGravity*
NorthEastGravity   *SouthEastGravity*
WestGravity   *StaticGravity*
CenterGravity

</div>

The win_gravity member is set to the window's window gravity and can be one of the following:

>   *UnmapGravity*   *EastGravity*
>   *NorthWestGravity*   *SouthWestGravity*
>   *NorthGravity*   *SouthGravity*
>   *NorthEastGravity*   *SouthEastGravity*
>   *WestGravity*   *StaticGravity*
>   *CenterGravity*

For additional information on gravity, see Section 4.3.

The backing_store member is set to indicate how the X server should maintain the contents of a window and can be *WhenMapped*, *Always* or *NotUseful*. The backing_planes member is set to indicate (with bits set to 1) which bit planes of the window hold dynamic data that must be preserved in backing_stores and during save_unders. The backing_pixel member is set to indicate what values to use for planes not set in backing_planes.

The save_under member is set to *True* or *False*. The colormap member is set to the colormap for the specified window and can be a colormap ID or *None*. The map_installed member is set to indicate whether the colormap is currently installed and can be *True* or *False*. The map_state member is set to indicate the state of the window and can be *IsUnmapped*, *IsUnviewable* or *IsViewable*. *IsUnviewable* is used if the window is mapped but some ancestor is unmapped.

The all_event_masks member is set to the bitwise inclusive OR of all event masks selected on the window by all clients. The your_event_mask member is set to the bitwise inclusive OR of all event masks selected by the querying client. The do_not_propagate_mask member is set to the bitwise inclusive OR of the set of events that should not propagate.

The override_redirect member is set to indicate whether this window overrides structure control facilities and can be *True* or *False*. Window manager clients should ignore the window if this member is *True*.

The screen member is set to a screen pointer that gives you a back pointer to the correct screen. This makes it easier to obtain the screen information without having to loop over the root window fields to see which field matches.

XGetWindowAttributes can generate *BadDrawable* and *BadWindow* errors.

To obtain the current geometry of a given drawable, use *XGetGeometry*.

```
Status XGetGeometry(display, d, root_return,
          x_return, y_return, width_return,
          height_return, border_width_return, depth_return)
     Display *display;
     Drawable d;
     Window *root_return;
     int *x_return, *y_return;
     unsigned int *width_return, *height_return;
     unsigned int *border_width_return;
     unsigned int *depth_return;
```

display   Specifies the connection to the X server.

d   Specifies the drawable, which can be a window or a pixmap.

root_return   Returns the root window.

x_return
y_return   Return the x and y coordinates that define the location of the drawable. For a window, these coordinates specify the upper-left outer corner relative to its parent's origin. For pixmaps, these coordinates are always zero.

width_return
height_return   Return the drawable's dimensions (width and height). For a window, these dimensions specify the inside size, not including the border.

border_width_return
                      Returns the border width in pixels. If the drawable is a pixmap, it returns zero.

depth_return   Returns the depth of the drawable (bits per pixel for the object).

The *XGetGeometry* function returns the root window and the current geometry of the drawable. The geometry of the drawable includes the x and y coordinates, width and height, border width and depth. These are described in the argument list. It is legal to pass to this function a window whose class is *InputOnly*.

XGetGeometry can generate a *BadDrawable* error.

5.2   Translating Screen Coordinates

Applications sometimes need to perform a coordinate transformation from the coordinate space of one window to another window or need to determine which window the pointing device is in. *XTranslateCoordinates* and *XQueryPointer* fulfill these needs (and avoid any race conditions) by asking the X server to perform these operations.

To translate a coordinate in one window to the coordinate space of another window, use *XTranslateCoordinates*.

```
Bool XTranslateCoordinates(display, src_w, dest_w,
        src_x, src_y, dest_x_return,
        dest_y_return, child_return)
    Display *display;
    Window src_w, dest_w;
    int src_x, src_y;
    int *dest_x_return, *dest_y_return;
    Window *child_return;
```

display   Specifies the connection to the X server.

src_w   Specifies the source window.

dest_w   Specifies the destination window.

src_x
src_y   Specify the x and y coordinates within the source window.

dest_x_return
dest_y_return   Return the x and y coordinates within the destination window.

child_return   Returns the child if the coordinates are contained in a mapped child of the destination window.

If *XTranslateCoordinates* returns *True*, it takes the src_x and src_y coordinates relative to the source window's origin and returns these coordinates to dest_x_return and dest_y_return relative to the destination window's origin. If *XTranslateCoordinates* returns *False*, src_w and dest_w are on different screens, and dest_x_return and dest_y_return are zero. If the coordinates are contained in a mapped child of dest_w, that child is returned to child_return. Otherwise, child_return is set to *None*.

XTranslateCoordinates can generate a *BadWindow* error.

To obtain the screen coordinates of the pointer, or to determine the pointer coordinates relative to a specified window, use *XQueryPointer*.

```
Bool XQueryPointer(display, w, root_return,
        child_return, root_x_return, root_y_return,
        win_x_return, win_y_return, mask_return)
    Display *display;
    Window w;
    Window *root_return, *child_return;
    int *root_x_return, *root_y_return;
    int *win_x_return, *win_y_return;
    unsigned int *mask_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

root_return   Returns the root window that the pointer is in.

child_return   Returns the child window that the pointer is located in, if any.

root_x_return
root_y_return   Return the pointer coordinates relative to the root window's origin.

win_x_return
win_y_return   Return the pointer coordinates relative to the specified window.

mask_return   Returns the current state of the modifier keys and pointer buttons.

The *XQueryPointer* function returns the root window the pointer is logically on and the pointer coordinates relative to the root window's origin. If *XQueryPointer* returns *False*, the pointer is not on the same screen as the specified window, and *XQueryPointer* returns *None* to child_return and zero to win_x_return and win_y_return. If *XQueryPointer* returns *True*, the pointer coordinates returned to win_x_return and win_y_return are relative to the origin of the specified window. In this case, *XQueryPointer* returns the child that contains the pointer, if any, or else *None* to child_return.

XQueryPointer returns the current logical state of the keyboard buttons and the modifier keys in mask_return. It sets mask_return to the bitwise inclusive OR of one or more of the button or modifier key bitmasks to match the current state of the mouse buttons and the modifier keys.

Note that the logical state of a device (as seen through Xlib) may lag the physical state if device event processing is frozen (see Section 13.1).

XQueryPointer can generate a *BadWindow* error.

5.3   **Properties and Atoms**

A property is a collection of named, typed data. The window system has a set of predefined properties (for example, the name of a window, size hints, and so on), and users can define any other arbitrary information and associate it with windows. Each property has a name, which is an ISO Latin-1 string. For each named property, a unique identifier (atom) is associated with it. A property also has a type; for example, string or integer. These types are also indicated using atoms, so arbitrary new types can be defined. Data of only one type may be associated with a single property name. Clients can store and retrieve properties associated with windows. For efficiency reasons, an atom is used rather than a character string. *XInternAtom* can be used to obtain the atom for property names.

A property is also stored in one of several possible formats. The X server can store the information as 8-bit quantities, 16-bit quantities or 32-bit quantities. This permits the X server to present the data in the byte order that the client expects.

Note: If you define further properties of complex type, you must encode and decode them yourself. These functions must be carefully written if they are to be portable. For further information about how to write a library extension, Appendix C.

The type of a property is defined by an atom, which allows for arbitrary extension in this type scheme.

Certain property names are predefined in the server for commonly used functions. The atoms for these properties are defined in <**X11/Xatom.h**>. To avoid name clashes with user symbols, the *#define* name for each atom has the XA_ prefix. For definitions of these properties, see Section 5.3. For an explanation of the functions that let you get and set much of the information stored in these predefined properties, see Chapter 15.

The core protocol imposes no semantics on these property names, but semantics are specified in other X Consortium standards, such as the **ICCCM** specification and the **XLFD** specification.

You can use properties to communicate other information between applications. The functions described in this section let you define new properties and get the unique atom IDs in your applications.

Although any particular atom can have some client interpretation within each of the name spaces, atoms occur in five distinct name spaces within the protocol:

- selections
- property names
- property types
- font properties
- type of a *ClientMessage* event (none are built into the X server).

The built-in selection property names are:

PRIMARY
SECONDARY

The built-in property names are:

```
CUT_BUFFER0            RESOURCE_MANAGER
CUT_BUFFER1            WM_CLASS
CUT_BUFFER2            WM_CLIENT_MACHINE
CUT_BUFFER3            WM_COLORMAP_WINDOWS
CUT_BUFFER4            WM_COMMAND
CUT_BUFFER5            WM_HINTS
CUT_BUFFER6            WM_ICON_NAME
CUT_BUFFER7            WM_ICON_SIZE
RGB_BEST_MAP           WM_NAME
RGB_BLUE_MAP           WM_NORMAL_HINTS
RGB_DEFAULT_MAP        WM_PROTOCOLS
RGB_GRAY_MAP           WM_STATE
RGB_GREEN_MAP          WM_TRANSIENT_FOR
RGB_RED_MAP
```

The built-in property types are:

```
ARC                    POINT
ATOM                   RGB_COLOR_MAP
BITMAP                 RECTANGLE
CARDINAL               STRING
COLORMAP               VISUALID
CURSOR                 WINDOW
DRAWABLE               WM_HINTS
FONT                   WM_SIZE_HINTS
INTEGER
PIXMAP
```

The built-in font property names are:

```
MIN_SPACE              STRIKEOUT_DESCENT
NORM_SPACE             STRIKEOUT_ASCENT
MAX_SPACE              ITALIC_ANGLE
END_SPACE              X_HEIGHT
SUPERSCRIPT_X          QUAD_WIDTH
SUPERSCRIPT_Y          WEIGHT
SUBSCRIPT_X            POINT_SIZE
SUBSCRIPT_Y            RESOLUTION
UNDERLINE_POSITION     COPYRIGHT
UNDERLINE_THICKNESS    NOTICE
FONT_NAME              FAMILY_NAME
FULL_NAME              CAP_HEIGHT
```

For further information about font properties, see Section 9.5.

To return an atom for a given name, use *XInternAtom*.

```
Atom XInternAtom(display, atom_name, only_if_exists)
      Display *display;
      char *atom_name;
      Bool only_if_exists;
```

display   Specifies the connection to the X server.

atom_name   Specifies the name associated with the atom you want returned.

only_if_exists   Specifies a Boolean value that indicates whether *XInternAtom* creates the atom.

The *XInternAtom* function returns the atom identifier associated with the specified atom_name string. If only_if_exists is *False*, the atom is created if it does not exist. Therefore, *XInternAtom* can return *None*. If the atom name is not in the Host Portable Character Encoding the result is implementation-dependent. Case matters; the strings *thing*, *Thing* and *thinG* all designate different atoms. The atom will remain defined even after the client's connection closes. It will become undefined only when the last connection to the X server closes.

XInternAtom can generate *BadAlloc* and *BadValue* errors.

To return a name for a given atom identifier, use *XGetAtomName*.

```
char *XGetAtomName(display, atom)
      Display *display;
      Atom atom;
```

display   Specifies the connection to the X server.

atom   Specifies the atom for the property name you want returned.

The *XGetAtomName* function returns the name associated with the specified atom. If the data returned by the server is in the Latin Portable Character Encoding, then the returned string is in the Host Portable Character Encoding. Otherwise, the result is implementation-dependent. To free the resulting string, call *XFree*.

XGetAtomName can generate a *BadAtom* error.

5.4   Obtaining and Changing Window Properties

You can attach a property list to every window. Each property has a name, a type, and a value (see Section 5.3). The value is an array of 8-bit, 16-bit, or 32-bit quantities, whose interpretation is left to the clients.

Xlib provides functions that you can use to obtain, change, update, or interchange window properties. In addition, Xlib provides other utility functions for inter-client communication (see Chapter 15).

To obtain the type, format, and value of a property of a given window, use *XGetWindowProperty*.

```
int XGetWindowProperty(display, w, property,
        long_offset, long_length, delete, req_type,
        actual_type_return, actual_format_return,
        nitems_return, bytes_after_return, prop_return)
    Display *display;
    Window w;
    Atom property;
    long long_offset, long_length;
    Bool delete;
    Atom req_type;
    Atom *actual_type_return;
    int *actual_format_return;
    unsigned long *nitems_return;
    unsigned long *bytes_after_return;
    unsigned char **prop_return;
```

display   Specifies the connection to the X server.

w   Specifies the window whose property you want to obtain.

property   Specifies the property name.

long_offset   Specifies the offset in the specified property (in 32-bit quantities) where the data is to be retrieved.

long_length   Specifies the length in 32-bit multiples of the data to be retrieved.

delete   Specifies a Boolean value that determines whether the property is deleted.

req_type   Specifies the atom identifier associated with the property type or *AnyPropertyType*.

actual_type_return
                     Returns the atom identifier that defines the actual type of the property.

actual_format_return
                     Returns the actual format of the property.

nitems_return   Returns the actual number of 8-bit, 16-bit, or 32-bit items stored in the prop_return data.

bytes_after_return Returns the number of bytes remaining to be read in the property if a partial read was performed.

prop_return   Returns the data in the specified format.

The *XGetWindowProperty* function returns the actual type of the property; the actual format of the property; the number of 8-bit, 16-bit or 32-bit items transferred; the number of bytes remaining to be read in the property; and a pointer to the data actually returned.

XGetWindowProperty sets the return arguments as follows:

- If the specified property does not exist for the specified window, *XGetWindowProperty* returns *None* to actual_type_return and the value zero to actual_format_return and bytes_after_return. The nitems_return argument is empty. In this case, the delete argument is ignored.

- If the specified property exists but its type does not match the specified type, *XGetWindowProperty* returns the actual property type to actual_type_return, the actual property format (never zero) to actual_format_return, and the property length in bytes (even if the actual_format_return is 16 or 32) to bytes_after_return. It also ignores the delete argument. The nitems_return argument is empty.

- If the specified property exists and either you assign *AnyPropertyType* to the req_type argument or the specified type matches the actual property type, *XGetWindowProperty* returns the actual property type to actual_type_return and the actual property format (never zero) to actual_format_return. It also returns a value to bytes_after_return and nitems_return, by defining the following values:

  N   Actual length of the stored property in bytes (even if the format is 16 or 32).

  I   4 * long_offset

  T   N - I

  L   MINIMUM(T, 4 * long_length)

  A   N - (I + L)

The returned value starts at byte index I in the property (indexing from zero), and its length in bytes is L. If the value for long_offset causes L to be negative, a *BadValue* error results. The value of bytes_after_return is A, giving the number of trailing unread bytes in the stored property.

XGetWindowProperty always allocates one extra byte in prop_return (even if the property is zero-length) and sets it to ASCII null so that simple properties consisting of characters do not have to be copied into yet another string before use. If delete is *True* and bytes_after_return is zero, *XGetWindowProperty* deletes the property from the window and generates a *PropertyNotify* event on the window.

The function returns *Success* if it executes successfully. To free the resulting data, use *XFree*.

XGetWindowProperty can generate *BadAtom*, *BadValue* and *BadWindow* errors.

To obtain a given window's property list, use *XListProperties*.

```
Atom *XListProperties(display, w, num_prop_return)
      Display *display;
      Window w;
      int *num_prop_return;
```

display   Specifies the connection to the X server.

w   Specifies the window whose property list you want to obtain.

num_prop_return   Returns the length of the properties array.

The *XListProperties* function returns a pointer to an array of atom properties that are defined for the specified window or returns NULL if no properties were found. To free the memory allocated by this function, use *XFree*.

XListProperties can generate a *BadWindow* error.

To change a property of a given window, use *XChangeProperty*.

```
XChangeProperty(display, w, property, type,
        format, mode, data, nelements)
    Display *display;
    Window w;
    Atom property, type;
    int format;
    int mode;
    unsigned char *data;
    int nelements;
```

display   Specifies the connection to the X server.

w   Specifies the window whose property you want to change.

property   Specifies the property name.

type   Specifies the type of the property. The X server does not interpret the type but
                   simply passes it back to an application that later calls *XGetWindowProperty*.

format   Specifies whether the data should be viewed as a list of 8-bit, 16-bit or 32-bit
                   quantities. Possible values are 8, 16 and 32. This information allows the X
                   server to correctly perform byte-swap operations as necessary. If the format is
                   16-bit or 32-bit, you must explicitly cast your data pointer to an (unsigned
                   char *) in the call to *XChangeProperty*.

mode   Specifies the mode of the operation. You can pass *PropModeReplace*,
                   PropModePrepend or *PropModeAppend*.

data   Specifies the property data.

nelements   Specifies the number of elements of the specified data format.

The *XChangeProperty* function alters the property for the specified window and causes the X
server to generate a *PropertyNotify* event on that window. *XChangeProperty* performs the
following:

- If mode is *PropModeReplace*, *XChangeProperty* discards the previous property value and stores
  the new data.

- If mode is *PropModePrepend* or *PropModeAppend*, *XChangeProperty* inserts the specified data
  before the beginning of the existing data or onto the end of the existing data, respectively.
  The type and format must match the existing property value, or a *BadMatch* error results. If
  the property is undefined, it is treated as defined with the correct type and format with zero-
  length data.

The lifetime of a property is not tied to the storing client. Properties remain until explicitly
deleted, until the window is destroyed, or until the server resets. For a discussion of what
happens when the connection to the X server is closed, see Section 3.6. The maximum size of a
property is server dependent and can vary dynamically depending on the amount of memory
the server has available. (If there is insufficient space, a *BadAlloc* error results.)

XChangeProperty can generate *BadAlloc*, *BadAtom*, *BadMatch*, *BadValue* and *BadWindow* errors.

To rotate a window's property list, use *XRotateWindowProperties*.

```
XRotateWindowProperties(display, w, properties,
        num_prop, npositions)
     Display *display;
     Window w;
     Atom properties[];
     int num_prop;
     int npositions;
```

display   Specifies the connection to the X server.

w   Specifies the window.

properties   Specifies the array of properties that are to be rotated.

num_prop   Specifies the length of the properties array.

npositions   Specifies the rotation amount.

The *XRotateWindowProperties* function allows you to rotate properties on a window and causes the X server to generate *PropertyNotify* events. If the property names in the properties array are viewed as being numbered starting from zero and if there are num_prop property names in the list, then the value associated with property name I becomes the value associated with property name (I + npositions) mod N for all I from zero to N −1. The effect is to rotate the states by npositions places around the virtual ring of property names (right for positive npositions, left for negative npositions). If npositions mod N is non-zero, the X server generates a *PropertyNotify* event for each property in the order that they are listed in the array. If an atom occurs more than once in the list or no property with that name is defined for the window, a *BadMatch* error results. If a *BadAtom* or *BadMatch* error results, no properties are changed.

XRotateWindowProperties can generate *BadAtom*, *BadMatch* and *BadWindow* errors.

To delete a property on a given window, use *XDeleteProperty*.

```
XDeleteProperty(display, w, property)
     Display *display;
     Window w;
     Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window whose property you want to delete.

property   Specifies the property name.

The *XDeleteProperty* function deletes the specified property only if the property was defined on the specified window and causes the X server to generate a *PropertyNotify* event on the window unless the property does not exist.

XDeleteProperty can generate *BadAtom* and *BadWindow* errors.

5.5   Selections

Selections are one method used by applications to exchange data. By using the property mechanism, applications can exchange data of arbitrary types and can negotiate the type of the data. A selection can be thought of as an indirect property with a dynamic type. That is, rather than having the property stored in the X server, the property is maintained by some client (the owner). A selection is global in nature (considered to belong to the user but be maintained by clients) rather than being private to a particular window subhierarchy or a particular set of clients.

Xlib provides functions that you can use to set, get or request conversion of selections. This allows applications to implement the notion of current selection, which requires that notification be sent to applications when they no longer own the selection. Applications that support selection often highlight the current selection and so must be informed when another application has acquired the selection so that they can unhighlight the selection.

When a client asks for the contents of a selection, it specifies a selection target type. This target type can be used to control the transmitted representation of the contents. For example, if the selection is ''the last thing the user clicked on'' and that is currently an image, then the target type might specify whether the contents of the image should be sent in XY format or Z format.

The target type can also be used to control the class of contents transmitted; for example, asking for the ''looks'' (fonts, line spacing, indentation, and so forth) of a paragraph selection, not the text of the paragraph. The target type can also be used for other purposes. The protocol does not constrain the semantics.

To set the selection owner, use *XSetSelectionOwner*.

```
XSetSelectionOwner(display, selection, owner, time)
      Display *display;
      Atom selection;
      Window owner;
      Time time;
```

display   Specifies the connection to the X server.

selection   Specifies the selection atom.

owner   Specifies the owner of the specified selection atom. You can pass a window or *None*.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XSetSelectionOwner* function changes the owner and last-change time for the specified selection and has no effect if the specified time is earlier than the current last-change time of the specified selection or is later than the current X server time. Otherwise, the last-change time is set to the specified time, with *CurrentTime* replaced by the current server time. If the owner window is specified as *None*, then the owner of the selection becomes *None* (that is, no owner). Otherwise, the owner of the selection becomes the client executing the request.

If the new owner (whether a client or *None*) is not the same as the current owner of the selection and the current owner is not *None*, the current owner is sent a *SelectionClear* event. If the client that is the owner of a selection is later terminated (that is, its connection is closed) or if the owner window it has specified in the request is later destroyed, the owner of the selection automatically reverts to *None*, but the last-change time is not affected. The selection atom is uninterpreted by the X server. *XGetSelectionOwner* returns the owner window, which is reported in *SelectionRequest* and *SelectionClear* events. Selections are global to the X server.

XSetSelectionOwner can generate *BadAtom* and *BadWindow* errors.

To return the selection owner, use *XGetSelectionOwner*.

```
Window XGetSelectionOwner(display, selection)
      Display *display;
      Atom selection;
```

display   Specifies the connection to the X server.

selection   Specifies the selection atom whose owner you want returned.

The *XGetSelectionOwner* function returns the window ID associated with the window that currently owns the specified selection. If no selection was specified, the function returns the constant *None*. If *None* is returned, there is no owner for the selection.

XGetSelectionOwner can generate a *BadAtom* error.

To request conversion of a selection, use *XConvertSelection*.

```
XConvertSelection(display, selection, target,
        property, requestor, time)
      Display *display;
      Atom selection, target;
      Atom property;
      Window requestor;
      Time time;
```

display   Specifies the connection to the X server.

selection   Specifies the selection atom.

target   Specifies the target atom.

property   Specifies the property name. You also can pass *None*.

requestor   Specifies the requestor.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

XConvertSelection requests that the specified selection be converted to the specified target type:

- If the specified selection has an owner, the X server sends a *SelectionRequest* event to that owner.

- If no owner for the specified selection exists, the X server generates a *SelectionNotify* event to the requestor with property *None*.

The arguments are passed on unchanged in either of the events. There are two predefined selection atoms: PRIMARY and SECONDARY.

XConvertSelection can generate *BadAtom* and *BadWindow* errors.

Pixmap and Cursor Functions

Once you have connected to an X server, you can use the Xlib functions to:

- create and free pixmaps
- create, recolor and free cursors.

6.1   Creating and Freeing Pixmaps

Pixmaps can only be used on the screen on which they were created. Pixmaps are off-screen resources that are used for various operations; for example, defining cursors as tiling patterns or as the source for certain raster operations. Most graphics requests can operate either on a window or on a pixmap. A bitmap is a single bit-plane pixmap.

To create a pixmap of a given size, use *XCreatePixmap*.

```
Pixmap XCreatePixmap(display, d, width, height, depth)
      Display *display;
      Drawable d;
      unsigned int width, height;
      unsigned int depth;
```

display   Specifies the connection to the X server.

d   Specifies which screen the pixmap is created on.

width
height   Specify the width and height, which define the dimensions of the pixmap.

depth   Specifies the depth of the pixmap.

The *XCreatePixmap* function creates a pixmap of the width, height, and depth you specified and returns a pixmap ID that identifies it. It is valid to pass an *InputOnly* window to the drawable argument. The width and height arguments must be non-zero, or a *BadValue* error results. The depth argument must be one of the depths supported by the screen of the specified drawable, or a *BadValue* error results.

The server uses the specified drawable to determine on which screen to create the pixmap. The pixmap can be used only on this screen and only with other drawables of the same depth (see *XCopyPlane* for an exception to this rule). The initial contents of the pixmap are undefined.

XCreatePixmap can generate *BadAlloc*, *BadDrawable* and *BadValue* errors.

To free all storage associated with a specified pixmap, use *XFreePixmap*.

```
XFreePixmap(display, pixmap)
      Display *display;
      Pixmap pixmap;
```

display   Specifies the connection to the X server.

pixmap   Specifies the pixmap.

The *XFreePixmap* function first deletes the association between the pixmap ID and the pixmap. Then, the X server frees the pixmap storage when there are no references to it. The pixmap should never be referenced again.

XFreePixmap can generate a *BadPixmap* error.

6.2   Creating, Recoloring and Freeing Cursors

Each window can have a different cursor defined for it. Whenever the pointer is in a visible window, it is set to the cursor defined for that window. If no cursor was defined for that window, the cursor is the one defined for the parent window.

From X's perspective, a cursor consists of a cursor source, mask, colors and a hotspot. The mask pixmap determines the shape of the cursor and must be a depth of one. The source pixmap must have a depth of one, and the colors determine the colors of the source. The hotspot defines the point on the cursor that is reported when a pointer event occurs. There may be limitations imposed by the hardware on cursors as to size and whether a mask is implemented. *XQueryBestCursor* can be used to find out what sizes are possible. There is a standard font for creating cursors, but Xlib provides functions that you can use to create cursors from an arbitrary font, or from bitmaps.

To create a cursor from the standard cursor font, use *XCreateFontCursor*.

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

Cursor XCreateFontCursor(display, shape)
      Display *display;
      unsigned int shape;
```

display   Specifies the connection to the X server.

shape   Specifies the shape of the cursor.

X provides a set of standard cursor shapes in a special font named cursor. Applications are encouraged to use this interface for their cursors because the font can be customized for the individual display type. The shape argument specifies which glyph of the standard fonts to use.

The hotspot comes from the information stored in the cursor font. The initial colors of a cursor are a black foreground and a white background (see *XRecolorCursor*). For further information about cursor shapes, see Appendix B.

XCreateFontCursor can generate *BadAlloc* and *BadValue* errors.

To create a cursor from font glyphs, use *XCreateGlyphCursor*.

```
Cursor XCreateGlyphCursor(display, source_font, mask_font,
         source_char, mask_char, foreground_color,
         background_color)
      Display *display;
      Font source_font, mask_font;
      unsigned int source_char, mask_char;
      XColor *foreground_color;
      XColor *background_color;
```

display   Specifies the connection to the X server.

source_font   Specifies the font for the source glyph.

mask_font   Specifies the font for the mask glyph or *None*.

source_char   Specifies the character glyph for the source.

mask_char   Specifies the glyph character for the mask.

foreground_color Specifies the RGB values for the foreground of the source.

background_color   Specifies the RGB values for the background of the source.

The *XCreateGlyphCursor* function is similar to *XCreatePixmapCursor* except that the source and mask bitmaps are obtained from the specified font glyphs. The source_char must be a defined glyph in source_font, or a *BadValue* error results. If mask_font is given, mask_char must be a defined glyph in mask_font, or a *BadValue* error results. The mask_font and character are optional. The origins of the source_char and mask_char (if defined) glyphs are positioned coincidently and define the hotspot. The source_char and mask_char need not have the same bounding box metrics, and there is no restriction on the placement of the hotspot relative to the bounding boxes. If no mask_char is given, all pixels of the source are displayed. You can free the fonts immediately by calling *XFreeFont* if no further explicit references to them are to be made.

For 2-byte matrix fonts, the 16-bit value should be formed with the byte1 member in the most-significant byte and the byte2 member in the least-significant byte.

XCreateGlyphCursor can generate *BadAlloc*, *BadFont* and *BadValue* errors.

To create a cursor from two bitmaps, use *XCreatePixmapCursor*.

```
Cursor XCreatePixmapCursor(display, source, mask,
        foreground_color, background_color, x, y)
    Display *display;
    Pixmap source;
    Pixmap mask;
    XColor *foreground_color;
    XColor *background_color;
    unsigned int x, y;
```

display   Specifies the connection to the X server.

source   Specifies the shape of the source cursor.

mask   Specifies the cursor's source bits to be displayed or *None*.

foreground_color   Specifies the RGB values for the foreground of the source.

background_color   Specifies the RGB values for the background of the source.

x
y   Specify the x and y coordinates, which indicate the hotspot relative to the source's origin.

The *XCreatePixmapCursor* function creates a cursor and returns the cursor ID associated with it. The foreground and background RGB values must be specified using foreground_color and background_color, even if the X server only has a *StaticGray* or *GrayScale* screen. The foreground color is used for the pixels set to 1 in the source, and the background color is used for the pixels set to 0. Both source and mask, if specified, must have depth one (or a *BadMatch* error results) but can have any root. The mask argument defines the shape of the cursor. The pixels set to 1 in the mask define which source pixels are displayed, and the pixels set to 0 define which pixels are ignored. If no mask is given, all pixels of the source are displayed. The mask, if present, must be the same size as the pixmap defined by the source argument, or a *BadMatch* error results. The hotspot must be a point within the source, or a *BadMatch* error results.

The components of the cursor can be transformed arbitrarily to meet display limitations. The pixmaps can be freed immediately if no further explicit references to them are to be made. Subsequent drawing in the source or mask pixmap has an undefined effect on the cursor. The X server might or might not make a copy of the pixmap.

XCreatePixmapCursor can generate *BadAlloc* and *BadPixmap* errors.

To determine useful cursor sizes, use *XQueryBestCursor*.

```
Status XQueryBestCursor(display, d, width, height,
        width_return, height_return)
    Display *display;
    Drawable d;
    unsigned int width, height;
    unsigned int *width_return, *height_return;
```

display   Specifies the connection to the X server.

d   Specifies the drawable, which indicates the screen.

width
height   Specify the width and height of the cursor that you want the size information for.

width_return
height_return   Return the best width and height that is closest to the specified width and height.

Some displays allow larger cursors than other displays. The *XQueryBestCursor* function provides a way to find out what size cursors are actually possible on the display. It returns the largest size that can be displayed. Applications should be prepared to use smaller cursors on displays that cannot support large ones.

XQueryBestCursor can generate a *BadDrawable* error.

To change the color of a given cursor, use *XRecolorCursor*.

```
XRecolorCursor(display, cursor, foreground_color,
        background_color)
    Display *display;
    Cursor cursor;
    XColor *foreground_color, *background_color;
```

display   Specifies the connection to the X server.

cursor   Specifies the cursor.

foreground_color Specifies the RGB values for the foreground of the source.

background_color Specifies the RGB values for the background of the source.

The *XRecolorCursor* function changes the color of the specified cursor, and if the cursor is being displayed on a screen, the change is visible immediately. Note that the pixel members of the *XColor* structures are ignored, only the RGB values are used.

XRecolorCursor can generate a *BadCursor* error.

To free (destroy) a given cursor, use *XFreeCursor*.

```
XFreeCursor(display, cursor)
    Display *display;
    Cursor cursor;
```

display   Specifies the connection to the X server.

cursor   Specifies the cursor.

The *XFreeCursor* function deletes the association between the cursor resource ID and the specified cursor. The cursor storage is freed when no other resource references it. The specified cursor ID should not be referred to again.

XFreeCursor can generate a *BadCursor* error.

Color Management Functions

Each X window always has an associated colormap that provides a level of indirection between pixel values and colors displayed on the screen. Xlib provides functions that you can use to manipulate a colormap. The X protocol defines colors using values in the RGB color space. The RGB color space is device-dependent; rendering an RGB value on differing output devices typically results in different colors. Xlib also provides a means for clients to specify color using device-independent color spaces, for consistent results across devices. Xlib supports device-independent color spaces derivable from the CIE XYZ color space. This includes the CIE XYZ, xyY, L*u*v*, and L*a*b* color spaces as well as the TekHVC color space.

This chapter discusses how to:

- create, copy and destroy a colormap
- specify colors by name or value
- allocate, modify and free color cells
- read entries in a colormap
- convert between color spaces
- control aspects of color conversion
- query the color gamut of a screen
- add new color spaces.

All functions, types and symbols in this chapter with the prefix "Xcms" are defined in **<X11/Xcms.h>**. The remaining functions and types are defined in **<X11/Xlib.h>**.

Functions in this chapter manipulate the representation of color on the screen. For each possible value that a pixel can take in a window, there is a color cell in the colormap. For example, if a window is 4 bits deep, pixel values 0 through 15 are defined. A colormap is a collection of color cells. A color cell consists of a triple of red, green and blue values. The hardware imposes limits on the number of significant bits in these values. As each pixel is read out of display memory, the pixel is looked up in a colormap. The RGB value of the cell determines what color is displayed on the screen. On a grayscale display with a black-and-white monitor the values are combined to determine the brightness on the screen.

Typically, an application allocates color cells or sets of color cells to obtain the desired colors. The client can allocate read-only cells, in which case the pixel values for these colors can be shared among multiple applications, and the RGB value of the cell cannot be changed. If the client allocates read/write cells, they are exclusively owned by the client, and the color associated with the pixel value may be changed at will. Cells must be allocated (and, if read/write, initialized with an RGB value) by a client to obtain desired colors; use of pixel value for an unallocated cell results in an undefined color.

Because colormaps are associated with windows, X supports displays with multiple colormaps and, indeed, different types of colormaps. If there are not sufficient colormap resources in the display, some windows will display in their true colors, and others will display with incorrect colors. A window manager usually controls which windows are displayed in their true colors if more than one colormap is required for the color resources the applications are using. At any time, there is a set of *installed* colormaps for a screen. Windows using one of the installed colormaps display with true colors, and windows using other colormaps generally display with

incorrect colors. The set of installed colormaps is controlled using *XInstallColormap* and *XUninstallColormap*.

Colormaps are local to a particular screen. Screens always have a default colormap, and programs typically allocate cells out of this colormap. You should not in general write applications that monopolize color resources. Although some hardware supports multiple colormaps installed at one time, many of the hardware displays built today support only a single installed colormap, so the primitives are written to encourage sharing of colormap entries between applications.

The *DefaultColormap* macro returns the default colormap. The *DefaultVisual* macro returns the default visual type for the specified screen. Possible visual types are *StaticGray*, *GrayScale*, *StaticColor*, *PseudoColor*, *TrueColor* or *DirectColor* (see Section 4.1).

7.1   Color Structures

Functions which operate only on RGB color space values use an *XColor* structure, which contains:

```
typedef struct {
    unsigned long pixel;            /* pixel value */
    unsigned short red, green, blue; /* rgb values */
    char flags;                     /* DoRed, DoGreen, DoBlue */
    char pad;
} XColor;
```

The red, green and blue values are always in the range 0 to 65535 inclusive, independent of the number of bits actually used in the display hardware. The server scales these values down to the range used by the hardware. Black is represented by (0,0,0), white is represented by (65535,65535,65535). In some functions, the flags member controls which of the red, green and blue members is used and can be the inclusive OR of zero or more of *DoRed*, *DoGreen* and *DoBlue*.

Functions which operate on all color space values use an *XcmsColor* structure. This structure contains a union of substructures, each supporting color specification encoding for a particular color space. Like the *XColor* structure, the *XcmsColor* structure contains pixel and color specification information (the spec member in the *XcmsColor* structure).

```
typedef unsigned long XcmsColorFormat; /* Color Specification Format */

typedef struct {
    union {
        XcmsRGB RGB;
        XcmsRGBi RGBi;
        XcmsCIEXYZ CIEXYZ;
        XcmsCIEuvY CIEuvY;
        XcmsCIExyY CIExyY;
        XcmsCIELab CIELab;
        XcmsCIELuv CIELuv;
        XcmsTekHVC TekHVC;
        XcmsPad Pad;
    } spec;
    XcmsColorFormat format;
    unsigned long pixel;
} XcmsColor;                          /* Xcms Color Structure */
```

Because the color specification can be encoded for the various color spaces, encoding for the spec member is identified by the format member, which is of type *XcmsColorFormat*. The following macros define standard formats.

```
#define    XcmsUndefinedFormat    0x00000000
#define    XcmsCIEXYZFormat       0x00000001    /* CIE XYZ */
#define    XcmsCIEuvYFormat       0x00000002    /* CIE u'v'Y */
#define    XcmsCIExyYFormat       0x00000003    /* CIE xyY */
#define    XcmsCIELabFormat       0x00000004    /* CIE L*a*b* */
#define    XcmsCIELuvFormat       0x00000005    /* CIE L*u*v* */
#define    XcmsTekHVCFormat       0x00000006    /* TekHVC */
#define    XcmsRGBFormat          0x80000000    /* RGB Device */
#define    XcmsRGBiFormat         0x80000001    /* RGB Intensity */
```

Note that formats for device-independent color spaces are distinguishable from those for device-dependent spaces by the 32nd bit. If this bit is set, it indicates that the color specification is in a device-dependent form; otherwise, it is in a device-independent form. If the 31st bit is set, this indicates that the color space has been added to Xlib at run-time (see Section 7.12.4). The format value for a color space added at run-time may be different each time the program is executed. If references to such a color space must be made outside the client (for example, storing a color specification in a file), then reference should be made by color space string prefix (see *XcmsFormatOfPrefix* and *XcmsPrefixOfFormat*).

Data types that describe the color specification encoding for the various color spaces are defined as follows:

```
typedef double XcmsFloat;

typedef struct {
    unsigned short red;      /* 0x0000 to 0xffff */
    unsigned short green;    /* 0x0000 to 0xffff */
    unsigned short blue;     /* 0x0000 to 0xffff */
} XcmsRGB;                    /* RGB Device */

typedef struct {
    XcmsFloat red;           /* 0.0 to 1.0 */
    XcmsFloat green;         /* 0.0 to 1.0 */
    XcmsFloat blue;          /* 0.0 to 1.0 */
} XcmsRGBi;                   /* RGB Intensity */

typedef struct {
    XcmsFloat X;
    XcmsFloat Y;             /* 0.0 to 1.0 */
    XcmsFloat Z;
} XcmsCIEXYZ;                 /* CIE XYZ */

typedef struct {
    XcmsFloat u_prime;       /* 0.0 to ˜0.6 */
    XcmsFloat v_prime;       /* 0.0 to ˜0.6 */
    XcmsFloat Y;             /* 0.0 to 1.0 */
} XcmsCIEuvY;                 /* CIE u'v'Y */

typedef struct {
    XcmsFloat x;             /* 0.0 to ˜.75 */
    XcmsFloat y;             /* 0.0 to ˜.85 */
    XcmsFloat Y;             /* 0.0 to 1.0 */
} XcmsCIExyY;                 /* CIE xyY */

typedef struct {
    XcmsFloat L_star;        /* 0.0 to 100.0 */
    XcmsFloat a_star;
    XcmsFloat b_star;
} XcmsCIELab;                 /* CIE L*a*b* */

typedef struct {
    XcmsFloat L_star;        /* 0.0 to 100.0 */
    XcmsFloat u_star;
    XcmsFloat v_star;
} XcmsCIELuv;                 /* CIE L*u*v* */
```

```
typedef struct {
    XcmsFloat H;                    /* 0.0 to 360.0 */
    XcmsFloat V;                    /* 0.0 to 100.0 */
    XcmsFloat C;                    /* 0.0 to 100.0 */
} XcmsTekHVC;                       /* TekHVC */

typedef struct {
    XcmsFloat pad0;
    XcmsFloat pad1;
    XcmsFloat pad2;
    XcmsFloat pad3;
} XcmsPad;                          /* four doubles */
```

The device-dependent formats provided allow color specification in:

- RGB Intensity (*XcmsRGBi*).

- Red, green and blue linear intensity values, floating point values from 0.0 to 1.0, where 1.0 indicates full intensity, 0.5 half intensity, and so on.

- RGB Device (*XcmsRGB*).

- Red, green and blue values appropriate for the specified output device. *XcmsRGB* values are of type unsigned short, scaled from 0 to 65535 inclusive, and are interchangeable with values the red, green and blue values in an *XColor* structure.

It is important to note that RGB Intensity values are not gamma corrected values. In contrast, RGB Device values generated as a result of converting color specifications are always gamma corrected, and RGB Device values acquired as a result of querying a colormap or passed in by the client are assumed by Xlib to be gamma corrected. The term ''RGB value'' in this manual always refers to an RGB Device value.

7.2   Color Strings

Xlib provides a mechanism for using string names for colors. A color string may either contain an abstract color name or a numerical color specification. Color strings are case-insensitive.

Color strings are used in the following functions:

- *XAllocNamedColor*

- *XcmsAllocNamedColor*

- *XLookupColor*

- *XcmsLookupColor*

- *XParseColor*

- *XStoreNamedColor*.

Xlib supports the use of abstract color names; for example, ''red'', ''blue''. A value for this abstract name is obtained by searching one or more color name databases. Xlib first searches zero or more client-side databases; the number, location and content of these databases is implementation-dependent, and might depend on the current locale. If the name is not found, Xlib then looks for the color in the X server's database. If the color name is not in the Host Portable Character Encoding the result is implementation-dependent.

A numerical color specification consists of a color space name and a set of values in the following syntax:

```
<color_space_name>:<value>/.../<value>
```

The following are examples of valid color strings.

```
"CIEXYZ:0.3227/0.28133/0.2493"
"RGBi:1.0/0.0/0.0"
"rgb:00/ff/00"
"CIELuv:50.0/0.0/0.0"
```

The syntax and semantics of numerical specifications are given for each standard color space in sections below.

7.2.1   RGB Device String Specification

An RGB Device specification is identified by the prefix ''rgb:'' and conforms to the following syntax:

```
rgb:<red>/<green>/<blue>

     <red>, <green>, <blue> := h | hh | hhh | hhhh
    h := single hexadecimal digits (case insignificant)
```

Note that *h* indicates the value scaled in 4 bits, *hh* the value scaled in 8 bits, *hhh* the value scaled in 12 bits, and *hhhh* the value scaled in 16 bits, respectively.

Typical examples are ''rgb:ea/75/52'' and ''rgb:ccc/320/320'', but mixed numbers of hex digits (''rgb:ff/a5/0'' and ''rgb:ccc/32/0'') are also allowed.

For backward compatibility, an older syntax for RGB Device is supported, but its continued use is not encouraged. The syntax is an initial sharp sign character followed by a numeric specification, in one of the following formats:

```
#RGB            (4 bits each)
#RRGGBB         (8 bits each)
#RRRGGGBBB      (12 bits each)
#RRRRGGGGBBBB   (16 bits each)
```

The R, G and B represent single hexadecimal digits. When fewer than 16 bits each are specified, they represent the most-significant bits of the value (unlike the ''rgb:'' syntax, in which values are scaled). For example, #3a7 is the same as #3000a0007000.

7.2.2   RGB Intensity String Specification

An RGB intensity specification is identified by the prefix ''rgbi:'' and conforms to the following syntax:

```
rgbi:<red>/<green>/<blue>
```

Note that red, green and blue are floating point values between 0.0 and 1.0, inclusive. The input format for these values is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an E or e followed by a possibly signed integer string.

7.2.3   Device-independent String Specifications

The standard device-independent string specifications have the following syntax:

```
CIEXYZ:<X>/<Y>/<Z>
CIEuvY:<u>/<v>/<Y>
CIExyY:<x>/<y>/<Y>
CIELab:<L>/<a>/<b>
CIELuv:<L>/<u>/<v>
TekHVC:<H>/<V>/<C>
```

All of the values (C, H, V, X, Y, Z, a, b, u, v, y, x) are floating point values. The syntax for these values is an optional '+' or '−' sign, a string of digits possibly containing a decimal point, and an optional exponent field consisting of an 'E' or 'e' followed by an optional '+' or '-' followed by a string of digits.

7.3   Color Conversion Contexts and Gamut Mapping

When Xlib converts device-independent color specifications into device-dependent specifications and *vice versa*, it uses knowledge about the color limitations of the screen hardware. This information, typically called the device profile, is available in a Color Conversion Context (hereafter referred to as the CCC).

Because a specified color may be outside the color gamut of the target screen and the white point associated with the color specification may differ from the white point inherent to the screen, Xlib applies gamut mapping when certain conditions are encountered:

- Gamut compression when conversion of device-independent color specifications to device-dependent color specification results in a color out of the target screen's gamut.

- White adjustment when the inherent white point of the screen differs from the white point assumed by the client.

Gamut handling methods are stored as callbacks in the CCC, which, in turn, are used by the color space conversion routines. Client data is also stored in the CCC for each callback. The CCC also contains the white point the client assumes to be associated with color specifications (that is, the Client White Point). The client can specify the gamut handling callbacks and client data, as well the Client White Point. Note that Xlib does not preclude the X client from performing other forms of gamut handling (for example, gamut expansion); however, direct support for gamut handling other than white adjustment and gamut compression is not provided by Xlib.

Associated with each colormap is an initial CCC transparently generated by Xlib. Therefore, when you specify a colormap as an argument to an Xlib function, you are indirectly specifying a CCC. There is a default CCC associated with each screen. Newly created CCCs inherit attributes from the default CCC, so the default CCC attributes can be modified to affect new CCCs.

Xcms functions in which gamut mapping can occur return *Status*, and have specific status values defined for them:

- *XcmsFailure* indicates that the function failed.

- *XcmsSuccess* indicates that the function succeeded. In addition, if the function performed any color conversion, the color (or colors) did not need to be compressed.

- *XcmsSuccessWithCompression* indicates the function performed color conversion, and at least one of the colors needed to be compressed. The gamut compression method is determined by the gamut compression procedure in the CCC that is specified directly as a function argument, or in the CCC indirectly specified by means of the colormap argument.

7.4   Creating, Copying and Destroying Colormaps

To create a colormap for a screen, use *XCreateColormap*.

```
Colormap XCreateColormap(display, w, visual, alloc)
      Display *display;
      Window w;
      Visual *visual;
      int alloc;
```

display   Specifies the connection to the X server.

w   Specifies the window on whose screen you want to create a colormap.

visual   Specifies a visual type supported on the screen. If the visual type is not one supported by the screen, a *BadMatch* error results.

alloc   Specifies the colormap entries to be allocated. You can pass *AllocNone* or *AllocAll*.

The *XCreateColormap* function creates a colormap of the specified visual type for the screen on which the specified window resides and returns the colormap ID associated with it. Note that the specified window is only used to determine the screen.

The initial values of the colormap entries are undefined for the visual classes *GrayScale*, *PseudoColor* and *DirectColor*. For *StaticGray*, *StaticColor* and *TrueColor*, the entries have defined values, but those values are specific to the visual and are not defined by X. For *StaticGray*, *StaticColor* and *TrueColor*, alloc must be *AllocNone*, or a *BadMatch* error results. For the other visual classes, if alloc is *AllocNone*, the colormap initially has no allocated entries, and clients can allocate them. For information about the visual types, see Section 4.1.

If alloc is *AllocAll*, the entire colormap is allocated writable. The initial values of all allocated entries are undefined. For *GrayScale* and *PseudoColor*, the effect is as if an *XAllocColorCells* call returned all pixel values from zero to N −1, where N is the colormap entries value in the specified visual. For *DirectColor*, the effect is as if an *XAllocColorPlanes* call returned a pixel value of zero and red_mask, green_mask and blue_mask values containing the same bits as the corresponding masks in the specified visual. However, in all cases, none of these entries can be freed by using *XFreeColors*.

XCreateColormap can generate *BadAlloc*, *BadMatch*, *BadValue* and *BadWindow* errors.

To create a new colormap when the allocation out of a previously shared colormap has failed because of resource exhaustion, use *XCopyColormapAndFree*.

```
Colormap XCopyColormapAndFree(display, colormap)
      Display *display;
      Colormap colormap;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

The *XCopyColormapAndFree* function creates a colormap of the same visual type and for the same screen as the specified colormap and returns the new colormap ID. It also moves all of the client's existing allocation from the specified colormap to the new colormap with their color values intact and their read-only or writable characteristics intact and frees those entries in the specified colormap. Color values in other entries in the new colormap are undefined. If the specified colormap was created by the client with alloc set to *AllocAll*, the new colormap is also created with *AllocAll*, all color values for all entries are copied from the specified colormap, and then all entries in the specified colormap are freed. If the specified colormap was not created by

the client with *AllocAll*, the allocations to be moved are all those pixels and planes that have been allocated by the client using *XAllocColor*, *XAllocNamedColor*, *XAllocColorCells* or *XAllocColorPlanes* and that have not been freed since they were allocated.

XCopyColormapAndFree can generate *BadAlloc* and *BadColor* errors.

To destroy a colormap, use *XFreeColormap*.

```
XFreeColormap(display, colormap)
      Display *display;
      Colormap colormap;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap that you want to destroy.

The *XFreeColormap* function deletes the association between the colormap resource ID and the colormap and frees the colormap storage. However, this function has no effect on the default colormap for a screen. If the specified colormap is an installed map for a screen, it is uninstalled (see *XUninstallColormap*). If the specified colormap is defined as the colormap for a window (by *XCreateWindow*, *XSetWindowColormap* or *XChangeWindowAttributes*), *XFreeColormap* changes the colormap associated with the window to *None* and generates a *ColormapNotify* event. X does not define the colors displayed for a window with a colormap of *None*.

XFreeColormap can generate a *BadColor* error.

7.5   **Mapping Color Names to Values**

To map a color name to an RGB value, use *XLookupColor*.

```
Status XLookupColor(display, colormap, color_name,
        exact_def_return, screen_def_return)
    Display *display;
    Colormap colormap;
    char *color_name;
    XColor *exact_def_return, *screen_def_return;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color_name   Specifies the color name string (for example, red) whose color definition structure you want returned.

exact_def_return   Returns the exact RGB values.

screen_def_return Returns the closest RGB values provided by the hardware.

The *XLookupColor* function looks up the string name of a color with respect to the screen associated with the specified colormap. It returns both the exact color values and the closest values provided by the screen with respect to the visual type of the specified colormap. If the color name is not in the Host Portable Character Encoding the result is implementation-dependent. Use of upper case or lower case does not matter. *XLookupColor* returns non-zero if the name is resolved, otherwise it returns zero.

XLookupColor can generate a *BadColor* error.

To map a color name to just the exact RGB value, use *XParseColor*.

```
Status XParseColor(display, colormap, spec, exact_def_return)
    Display *display;
    Colormap colormap;
    char *spec;
    XColor *exact_def_return;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

spec   Specifies the color name string; case is ignored.

exact_def_return   Returns the exact color value for later use and sets the *DoRed*, *DoGreen* and *DoBlue* flags.

The *XParseColor* function looks up the string name of a color with respect to the screen associated with the specified colormap. It returns the exact color value. If the color name is not in the Host Portable Character Encoding the result is implementation-dependent. Use of upper case or lower case does not matter. *XParseColor* returns non-zero if the name is resolved, otherwise it returns zero.

XParseColor can generate a *BadColor* error.

To map a color name to a value in an arbitrary color space, use *XcmsLookupColor*.

```
Status XcmsLookupColor(display, colormap, color_string,
        color_exact_return, color_screen_return, result_format)
    Display *display;
    Colormap colormap;
    char *color_string;
    XcmsColor *color_exact_return, *color_screen_return;
    XcmsColorFormat result_format;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color_string   Specifies the color string.

color_exact_return Returns the color specification parsed from the color string or parsed from the
                    corresponding string found in a color name database.

color_screen_return
                    Returns the color that can be reproduced on the *Screen*.

result_format   Specifies the color format for the returned color specifications
                    (color_screen_return and color_exact_return arguments). If format is
                    XcmsUndefinedFormat and the color string contains a numerical color
                    specification, the specification is returned in the format used in that numerical
                    color specification. If format is *XcmsUndefinedFormat* and the color string
                    contains a color name, the specification is returned in the format used to store
                    the color in the database.

The *XcmsLookupColor* function looks up the string name of a color with respect to the screen
associated with the specified colormap. It returns both the exact color values and the closest
values provided by the screen with respect to the visual type of the specified colormap. The
values are returned in the format specified by result_format. If the color name is not in the Host
Portable Character Encoding the result is implementation-dependent. Use of upper case or
lower case does not matter. *XcmsLookupColor* returns *XcmsSuccess* or
XcmsSuccessWithCompression if the name is resolved, otherwise it returns *XcmsFailure*. If
XcmsSuccessWithCompression is resturned, then the color specification in *color_screen_return* is the
result of gamut compression.

7.6   **Allocating and Freeing Color Cells**

There are two ways of allocating color cells: explicitly as read-only entries, one pixel value at a
time, or read/write, where you can allocate a number of color cells and planes simultaneously.
A read-only cell has its RGB value set by the server. Read/write cells do not have defined colors
initially; functions described in the next section must be used to store values into them.
Although it is possible for any client to store values into a read/write cell allocated by another
client, read/write cells normally should be considered private to the client that allocated them.

Read-only colormap cells are shared among clients. The server counts each allocation and free
of the cell by clients. When the last client frees a shared cell, the cell is finally deallocated. Note
that if a single client allocates the same read-only cell multiple times, the server counts each such
allocation, not just the first one.

To allocate a read-only color cell with an RGB value, use *XAllocColor*.

```
Status XAllocColor(display, colormap, screen_in_out)
      Display *display;
      Colormap colormap;
      XColor *screen_in_out;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

screen_in_out   Specifies and returns the values actually used in the colormap.

The *XAllocColor* function allocates a read-only colormap entry corresponding to the closest RGB
value supported by the hardware. *XAllocColor* returns the pixel value of the color closest to the
specified RGB elements supported by the hardware and returns the RGB value actually used.
The corresponding colormap cell is read-only. In addition, *XAllocColor* returns non-zero if it
succeeded or zero if it failed. Multiple clients that request the same effective RGB value can be
assigned the same read-only entry, thus allowing entries to be shared. When the last client
deallocates a shared cell, it is deallocated. *XAllocColor* does not use or affect the flags in the
XColor structure.

XAllocColor can generate a *BadColor* error.

To allocate a read-only color cell with a color in arbitrary format, use *XcmsAllocColor*.

```
Status XcmsAllocColor(display, colormap, color_in_out, result_format)
      Display *display;
      Colormap colormap;
      XcmsColor *color_in_out;
      XcmsColorFormat result_format;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color_in_out   Specifies the color to allocate and returns the pixel and color that is actually
                      used in the colormap.

result_format   Specifies the color format for the returned color specification.

The *XcmsAllocColor* function is similar to *XAllocColor* except the color can be specified in any
format. The *XcmsAllocColor* function ultimately calls *XAllocColor* to allocate a read-only color
cell (colormap entry) with the specified color. *XcmsAllocColor* first converts the color specified to
an RGB value and then passes this to *XAllocColor*. *XcmsAllocColor* returns the pixel value of the
color cell and the color specification actually allocated. This returned color specification is the

result of converting the RGB value returned by *XAllocColor* into the format specified with the result_format argument. If there is no interest in a returned color specification, unnecessary computation can be bypassed if result_format is set to *XcmsRGBFormat*. The corresponding colormap cell is read-only. If this routine returns *XcmsFailure*, the color_in_out color specification is left unchanged.

XcmsAllocColor can generate a *BadColor* error.

To allocate a read-only color cell using a color name, and return the closest color supported by the hardware in RGB format, use *XAllocNamedColor*.

```
Status XAllocNamedColor(display, colormap, color_name,
        screen_def_return, exact_def_return)
    Display *display;
    Colormap colormap;
    char *color_name;
    XColor *screen_def_return, *exact_def_return;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color_name   Specifies the color name string (for example, red) whose color definition structure you want returned.

screen_def_return   Returns the closest RGB values provided by the hardware.

exact_def_return   Returns the exact RGB values.

The *XAllocNamedColor* function looks up the named color with respect to the screen that is associated with the specified colormap. It returns both the exact database definition and the closest color supported by the screen. The allocated color cell is read-only. The pixel value is returned in screen_def_return. If the color name is not in the Host Portable Character Encoding the result is implementation-dependent. Use of upper case or lower case does not matter. *XLookupColor* returns non-zero if a cell is allocated, otherwise it returns zero.

XAllocNamedColor can generate a *BadColor* error.

To allocate a read-only color cell using a color name, and return the closest color supported by the hardware in an arbitrary format, use *XcmsAllocNamedColor*.

```
Status XcmsAllocNamedColor(display, colormap, color_string,
        result_format, color_screen_return, color_exact_return)
    Display *display;
    Colormap colormap;
    char *color_string;
    XcmsColorFormat result_format;
    XcmsColor *color_screen_return;
    XcmsColor *color_exact_return;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color_string   Specifies the color string whose color definition structure is to be returned.

result_format   Specifies the color format for the returned color specifications (color_screen_return and color_exact_return arguments). If format is *XcmsUndefinedFormat* and the color string contains a numerical color specification, the specification is returned in the format used in that numerical

color specification. If format is *XcmsUndefinedFormat* and the color string contains a color name, the specification is returned in the format used to store the color in the database.

color_screen_return

Returns the pixel value of the color cell and color specification that actually is stored for that cell.

color_exact_return Returns the color specification parsed from the color string or parsed from the corresponding string found in a color name database.

The *XcmsAllocNamedColor* function is similar to *XAllocNamedColor* except the color returned can be in any format specified. This function ultimately calls *XAllocColor* to allocate a read-only color cell with the color specified by a color string. The color string is parsed into an *XcmsColor* structure (see *XcmsLookupColor*), converted to an RGB value, then finally passed to the *XAllocColor*. If the color name is not in the Host Portable Character Encoding the result is implementation-dependent. Use of upper case or lower case does not matter.

This function returns both the color specification as a result of parsing (exact specification) and the actual color specification stored (screen specification). This screen specification is the result of converting the RGB value returned by *XAllocColor* into the format specified in result_format. If there is no interest in a returned color specification, unnecessary computation can be bypassed if result_format is set to *XcmsRGBFormat*.

XcmsAllocNamedColor can generate a *BadColor* error.

To allocate read/write color cell and color plane combinations for a *PseudoColor* model, use *XAllocColorCells*.

```
Status XAllocColorCells(display, colormap, contig,
        plane_masks_return, nplanes, pixels_return, npixels)
    Display *display;
    Colormap colormap;
    Bool contig;
    unsigned long plane_masks_return[];
    unsigned int nplanes;
    unsigned long pixels_return[];
    unsigned int npixels;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

contig   Specifies a Boolean value that indicates whether the planes must be contiguous.

plane_mask_return Returns an array of plane masks.

nplanes   Specifies the number of plane masks that are to be returned in the plane masks array.

pixels_return   Returns an array of pixel values.

npixels   Specifies the number of pixel values that are to be returned in the pixels_return array.

The *XAllocColorCells* function allocates read/write color cells. The number of colors must be positive and the number of planes non-negative, or a *BadValue* error results. If ncolors and nplanes are requested, then ncolors pixels and nplane plane masks are returned. No mask will have any bits set to 1 in common with any other mask or with any of the pixels. By ORing

together each pixel with zero or more masks, ncolors $* 2^{nplanes}$ distinct pixels can be produced. All of these are allocated writable by the request. For *GrayScale* or *PseudoColor*, each mask has exactly one bit set to 1. For *DirectColor*, each has exactly three bits set to 1. If contig is *True* and if all masks are ORed together, a single contiguous set of bits set to 1 will be formed for *GrayScale* or *PseudoColor* and three contiguous sets of bits set to 1 (one within each pixel subfield) for *DirectColor*. The RGB values of the allocated entries are undefined. *XAllocColorCells* returns non-zero if it succeeded or zero if it failed.

XAllocColorCells can generate *BadColor* and *BadValue* errors.

To allocate read/write color resources for a *DirectColor* model, use *XAllocColorPlanes*.

```
Status XAllocColorPlanes(display, colormap, contig,
        pixels_return, ncolors, nreds, ngreens,
        nblues, rmask_return, gmask_return, bmask_return)
    Display *display;
    Colormap colormap;
    Bool contig;
    unsigned long pixels_return[];
    int ncolors;
    int nreds, ngreens, nblues;
    unsigned long *rmask_return, *gmask_return, *bmask_return;
```

display	Specifies the connection to the X server.
colormap	Specifies the colormap.
contig	Specifies a Boolean value that indicates whether the planes must be contiguous.
pixels_return	Returns an array of pixel values. *XAllocColorPlanes* returns the pixel values in this array.
ncolors	Specifies the number of pixel values that are to be returned in the pixels_return array.
nreds *ngreens* *nblues*	Specify the number of red, green and blue planes. The value you pass must be non-negative.
rmask_return *gmask_return* *bmask_return*	Return bit masks for the red, green and blue planes.

The specified ncolors must be positive; and nreds, ngreens and nblues must be non-negative, or a *BadValue* error results. If ncolors colors, nreds reds, ngreens greens and nblues blues are requested, ncolors pixels are returned; and the masks have nreds, ngreens and nblues bits set to 1, respectively. If contig is *True*, each mask will have a contiguous set of bits set to 1. No mask will have any bits set to 1 in common with any other mask or with any of the pixels. For *DirectColor*, each mask will lie within the corresponding pixel subfield. By ORing together subsets of masks with each pixel value, ncolors $* 2^{(nreds+ngreens+nblues)}$ distinct pixel values can be produced. All of these are allocated by the request. However, in the colormap, there are only ncolors $* 2^{nreds}$ independent red entries, ncolors $* 2^{ngreens}$ independent green entries, and ncolors $* 2^{nblues}$ independent blue entries. This is true even for *PseudoColor*. When the colormap entry of a pixel value is changed (using *XStoreColors*, *XStoreColor* or *XStoreNamedColor*), the pixel is decomposed according to the masks, and the corresponding independent entries are updated.

XAllocColorPlanes returns non-zero if it succeeded or zero if it failed.

XAllocColorPlanes can generate *BadColor* and *BadValue* errors.

To free colormap cells, use *XFreeColors*.

```
XFreeColors(display, colormap, pixels, npixels, planes)
      Display *display;
      Colormap colormap;
      unsigned long pixels[];
      int npixels;
      unsigned long planes;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

pixels   Specifies an array of pixel values that map to the cells in the specified colormap.

npixels   Specifies the number of pixels.

planes   Specifies the planes you want to free.

The *XFreeColors* function frees the cells represented by pixels whose values are in the pixels array. The planes argument should not have any bits set to 1 in common with any of the pixels. The set of all pixels is produced by ORing together subsets of the planes argument with the pixels. The request frees all of these pixels that were allocated by the client (using *XAllocColor*, *XAllocNamedColor*, *XAllocColorCells* and *XAllocColorPlanes*). Note that freeing an individual pixel obtained from *XAllocColorPlanes* may not actually allow it to be reused until all of its related pixels are also freed. Similarly, a read-only entry is not actually freed until it has been freed by all clients, and if a client allocates the same read-only entry multiple times, it must free the entry that many times before the entry is actually freed.

All specified pixels that are allocated by the client in the colormap are freed, even if one or more pixels produce an error. If a specified pixel is not a valid index into the colormap, a *BadValue* error results. If a specified pixel is not allocated by the client (that is, is unallocated or is only allocated by another client), or if the colormap was created with all entries writable (by passing *AllocAll* to *XCreateColormap*), a *BadAccess* error results. If more than one pixel is in error, the one that gets reported is arbitrary.

XFreeColors can generate *BadAccess*, *BadColor* and *BadValue* errors.

7.7   Modifying and Querying Colormap Cells

To store an RGB value in a single colormap cell, use *XStoreColor*.

```
XStoreColor(display, colormap, color)
      Display *display;
      Colormap colormap;
      XColor *color;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color   Specifies the pixel and RGB values.

The *XStoreColor* function changes the colormap entry of the pixel value specified in the pixel member of the *XColor* structure. You specified this value in the pixel member of the *XColor* structure. This pixel value must be a read/write cell and a valid index into the colormap. If a specified pixel is not a valid index into the colormap, a *BadValue* error results. *XStoreColor* also changes the red, green and/or blue color components. You specify which color components are to be changed by setting *DoRed*, *DoGreen* and/or *DoBlue* in the flags member of the *XColor* structure. If the colormap is an installed map for its screen, the changes are visible immediately.

XStoreColor can generate *BadAccess*, *BadColor* and *BadValue* errors.

To store multiple RGB values into multiple colormap cells, use *XStoreColors*.

```
XStoreColors(display, colormap, color, ncolors)
      Display *display;
      Colormap colormap;
      XColor color[];
      int ncolors;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color   Specifies an array of color definition structures to be stored.

ncolors   Specifies the number of *XColor* structures in the color definition array.

The *XStoreColors* function changes the colormap entries of the pixel values specified in the pixel members of the *XColor* structures. You specify which color components are to be changed by setting *DoRed*, *DoGreen* and/or *DoBlue* in the flags member of the *XColor* structures. If the colormap is an installed map for its screen, the changes are visible immediately. *XStoreColors* changes the specified pixels if they are allocated writable in the colormap by any client, even if one or more pixels generates an error. If a specified pixel is not a valid index into the colormap, a *BadValue* error results. If a specified pixel either is unallocated or is allocated read-only, a *BadAccess* error results. If more than one pixel is in error, the one that gets reported is arbitrary.

XStoreColors can generate *BadAccess*, *BadColor* and *BadValue* errors.

To store a color of arbitrary format in a single colormap cell, use *XcmsStoreColor*.

```
Status XcmsStoreColor(display, colormap, color)
      Display *display;
      Colormap colormap;
      XcmsColor *color;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color   Specifies the color cell and the color to store. Values specified in this
                    XcmsColor structure remain unchanged upon return.

The *XcmsStoreColor* function converts the color specified in the *XcmsColor* structure into RGB
values and then uses this RGB specification in an *XColor* structure, whose three flags (*DoRed*,
DoGreen and *DoBlue*) are set, in a call to *XStoreColor* to change the color cell specified by the pixel
member of the *XcmsColor* structure. This pixel value must be a valid index for the specified
colormap, and the color cell specified by the pixel value must be a read/write cell. If the pixel
value is not a valid index, a *BadValue* error results. If the color cell is unallocated or is allocated
read-only, a *BadAccess* error results. If the colormap is an installed map for its screen, the
changes are visible immediately.

Note that *XStoreColor* has no return value; therefore, a *XcmsSuccess* return value from this
function indicates that the conversion to RGB succeeded and the call to *XStoreColor* was made.
To obtain the actual color stored, use *XcmsQueryColor*. Due to the screen's hardware limitations
or gamut compression, the color stored in the colormap may not be identical to the color
specified.

XcmsStoreColor can generate *BadAccess*, *BadColor* and *BadValue* errors.

To store multiple colors of arbitrary format into multiple colormap cells, use *XcmsStoreColors*.

```
Status XcmsStoreColors(display, colormap, colors,
        ncolors, compression_flags_return)
    Display *display;
    Colormap colormap;
    XcmsColor colors[];
    int ncolors;
    Bool compression_flags_return[];
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

colors   Specifies the color specification array of *XcmsColor* structures, each specifying
                    a color cell and the color to store in that cell. Values specified in the array
                    remain unchanged upon return.

ncolors   Specifies the number of *XcmsColor* structures in the color specification array.

compression_flags_return
                    Specifies an array of Boolean values for returning compression status. If a
                    non-NULL pointer is supplied, each element of the array is set to *True* if the
                    corresponding color was compressed, and *False* otherwise. Pass NULL if the
                    compression status is not useful.

The *XcmsStoreColors* function converts the colors specified in the array of *XcmsColor* structures
into RGB values and then uses these RGB specifications in an *XColor* structures, whose three
flags (*DoRed*, *DoGreen* and *DoBlue*) are set, in a call to *XStoreColors* to change the color cells
specified by the pixel member of the corresponding *XcmsColor* structure. Each pixel value must
be a valid index for the specified colormap, and the color cell specified by each pixel value must
be a read/write cell. If a pixel value is not a valid index, a *BadValue* error results. If a color cell is
unallocated or is allocated read-only, a *BadAccess* error results. If more than one pixel is in error,
the one that gets reported is arbitrary. If the colormap is an installed map for its screen, the
changes are visible immediately.

Note that *XStoreColors* has no return value; therefore, a *XcmsSuccess* return value from this function indicates that conversions to RGB succeeded and the call to *XStoreColors* was made. To obtain the actual colors stored, use *XcmsQueryColors*. Due to the screen's hardware limitations or gamut compression, the colors stored in the colormap may not be identical to the colors specified.

XcmsStoreColors can generate *BadAccess*, *BadColor* and *BadValue* errors.

To store a color specified by name in a single colormap cell, use *XStoreNamedColor*.

```
XStoreNamedColor(display, colormap, color, pixel, flags)
      Display *display;
      Colormap colormap;
      char *color;
      unsigned long pixel;
      int flags;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color   Specifies the color name string (for example, red).

pixel   Specifies the entry in the colormap.

flags   Specifies which red, green and blue components are set.

The *XStoreNamedColor* function looks up the named color with respect to the screen associated with the colormap and stores the result in the specified colormap. The pixel argument determines the entry in the colormap. The flags argument determines which of the red, green and blue components are set. You can set this member to the bitwise inclusive OR of the bits *DoRed*, *DoGreen* and *DoBlue*. If the color name is not in the Host Portable Character Encoding the result is implementation-dependent. Use of upper case or lower case does not matter. If the specified pixel is not a valid index into the colormap, a *BadValue* error results. If the specified pixel either is unallocated or is allocated read-only, a *BadAccess* error results.

XStoreNamedColor can generate *BadAccess*, *BadColor*, *BadName* and *BadValue* errors.

The *XQueryColor* and *XQueryColors* functions take pixel values in the pixel member of *XColor* structures, and store in the structures the RGB values for those pixels from the specified colormap. The values returned for an unallocated entry are undefined. These functions also set the flags member in the *XColor* structure to all three colors. If a pixel is not a valid index into the specified colormap, a *BadValue* error results. If more than one pixel is in error, the one that gets reported is arbitrary.

To query the RGB value of a single colormap cell, use *XQueryColor*.

```
XQueryColor(display, colormap, def_in_out)
      Display *display;
      Colormap colormap;
      XColor *def_in_out;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

def_in_out   Specifies and returns the RGB values for the pixel specified in the structure.

The *XQueryColor* function returns the current RGB value for the pixel in the *XColor* structure and sets the *DoRed*, *DoGreen* and *DoBlue* flags.

XQueryColor can generate *BadColor* and *BadValue* errors.

To query the RGB values of multiple colormap cells, use *XQueryColors*.

```
XQueryColors(display, colormap, defs_in_out, ncolors)
      Display *display;
      Colormap colormap;
      XColor defs_in_out[];
      int ncolors;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

defs_in_out   Specifies and returns an array of color definition structures for the pixel specified in the structure.

ncolors   Specifies the number of *XColor* structures in the color definition array.

The *XQueryColors* function returns the RGB value for each pixel in each *XColor* structure, and sets the *DoRed*, *DoGreen* and *DoBlue* flags in each structure.

XQueryColors can generate *BadColor* and *BadValue* errors.

To query the color of a single colormap cell in an arbitrary format, use *XcmsQueryColor*.

```
Status XcmsQueryColor(display, colormap, color_in_out, result_format)
      Display *display;
      Colormap colormap;
      XcmsColor *color_in_out;
      XcmsColorFormat result_format;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

color_in_out   Specifies the pixel member that indicates the color cell to query, and the color specification stored for the color cell is returned in this *XcmsColor* structure.

result_format   Specifies the color format for the returned color specification.

The *XcmsQueryColor* function obtains the RGB value for the pixel value in the pixel member of the specified *XcmsColor* structure, and then converts the value to the target format as specified by the result_format argument. If the pixel is not a valid index into the specified colormap, a *BadValue* error results.

XcmsQueryColor can generate *BadColor* and *BadValue* errors.

To query the color of multiple colormap cells in an arbitrary format, use *XcmsQueryColors*.

```
Status XcmsQueryColors(display, colormap, colors_in_out,
          ncolors, result_format)
      Display *display;
      Colormap colormap;
      XcmsColor colors_in_out[];
      unsigned int ncolors;
      XcmsColorFormat result_format;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

colors_in_out   Specifies an array of *XcmsColor* structures, each pixel member indicating the color cell to query. The color specifications for the color cells are returned in these structures.

ncolors   Specifies the number of *XcmsColor* structures in the color specification array.

result_format   Specifies the color format for the returned color specification.

The *XcmsQueryColors* function obtains the RGB values for pixel values in the pixel members of *XcmsColor* structures, and then converts the values to the target format as specified by the result_format argument. If a pixel is not a valid index into the specified colormap, a *BadValue* error results. If more than one pixel is in error, the one that gets reported is arbitrary.

XcmsQueryColors can generate *BadColor* and *BadValue* errors.

7.8   Color Conversion Context Functions

This section describes functions to create, modify and query Color Conversion Contexts.

Associated with each colormap is an initial CCC transparently generated by Xlib. Therefore, when you specify a colormap as an argument to a function, you are indirectly specifying a CCC. The CCC attributes that can be modified by the X client are:

- Client White Point

- gamut compression procedure and client data

- white point adjustment procedure and client data.

The initial values for these attributes are implementation-specific. The CCC attributes for subsequently created CCCs can be defined by changing the CCC attributes of the default CCC. There is a default CCC associated with each screen.

7.8.1   Getting and Setting the Color Conversion Context of a Colormap

To obtain the CCC associated with a colormap, use *XcmsCCCOfColormap*.

```
XcmsCCC XcmsCCCofColormap(display, colormap)
      Display *display;
      Colormap colormap;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

The *XcmsCCCofColormap* function returns the CCC associated with the specified colormap. Once obtained, the CCC attributes can be queried or modified. Unless the CCC associated with the specified colormap is changed with *XcmsSetCCCOfColormap*, this CCC is used when the specified colormap is used as an argument to color functions.

To change the CCC associated with a colormap, use *XcmsSetCCCOfColormap*.

```
XcmsCCC XcmsSetCCCOfColormap(display, colormap, ccc)
      Display *display;
      Colormap colormap;
      XcmsCCC ccc;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

ccc   Specifies the CCC.

The *XcmsSetCCCOfColormap* function changes the CCC associated with the specified colormap. It returns the CCC previously associated to the colormap. If they are not used again in the application, CCCs should be freed by calling *XcmsFreeCCC*.

7.8.2   Obtaining the Default Color Conversion Context

The default CCC attributes for subsequently created CCCs can be changed by changing the CCC attributes of the default CCC. A default CCC is associated with each screen.

To obtain the default CCC for a screen, use *XcmsDefaultCCC*.

```
XcmsCCC XcmsDefaultCCC(display, screen_number)
      Display *display;
      int screen_number;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

The *XcmsDefaultCCC* function returns the default CCC for the specified screen. Its visual is the default visual of the screen. Its initial gamut compression and white point adjustment procedures as well as the associated client data are implementation-specific.

7.8.3   Color Conversion Context Macros

Applications should not directly modify any part of the *XcmsCCC*. The following lists the C-language macros, their corresponding function equivalents that are for other language bindings, and what data they both can return.

```
DisplayOfCCC(ccc)
      XcmsCCC ccc;

Display *XcmsDisplayOfCCC(ccc)
      XcmsCCC ccc;
```

ccc   Specifies the CCC.

Both return the display associated with the specified CCC.

```
VisualOfCCC(ccc)
      XcmsCCC ccc;

Visual *XcmsVisualOfCCC(ccc)
      XcmsCCC ccc;
```

ccc   Specifies the CCC.

Both return the visual associated with the specified CCC.

```
ScreenNumberOfCCC(ccc)
      XcmsCCC ccc;

int XcmsScreenNumberOfCCC(ccc)
      XcmsCCC ccc;
```

ccc   Specifies the CCC.

Both return the number of the screen associated with the specified CCC.

```
ScreenWhitePointOfCCC(ccc)
      XcmsCCC ccc;

XcmsColor *XcmsScreenWhitePointOfCCC(ccc)
      XcmsCCC ccc;
```

ccc   Specifies the CCC.

Both return the white point of the screen associated with the specified CCC.

```
ClientWhitePointOfCCC(ccc)
      XcmsCCC ccc;


XcmsColor *XcmsClientWhitePointOfCCC(ccc)
      XcmsCCC ccc;
```

ccc   Specifies the CCC.

Both return the Client White Point of the specified CCC.

7.8.4   Modifying Attributes of a Color Conversion Context

To set the Client White Point in the CCC, use *XcmsSetWhitePoint*.

```
Status XcmsSetWhitePoint(ccc, color)
      XcmsCCC ccc;
      XcmsColor *color;
```

ccc   Specifies the CCC.

color   Specifies the new Client White Point.

The *XcmsSetWhitePoint* function changes the Client White Point in the specified CCC. Note that the pixel member is ignored and that the color specification is left unchanged upon return. The format for the new white point must be *XcmsCIEXYZFormat*, *XcmsCIEuvYFormat*, *XcmsCIExyYFormat* or *XcmsUndefinedFormat*. If *color* is NULL, this function sets the format component of the Client White Point specification to *XcmsUndefinedFormat*, indicating that the Client White Point is assumed to be the same as the Screen White Point.

To set the gamut compression procedure and corresponding client data in a specified CCC, use *XcmsSetCompressionProc*.

```
XcmsCompressionProc XcmsSetCompressionProc(ccc, compression_proc,
          client_data)
      XcmsCCC ccc;
      XcmsCompressionProc compression_proc;
      XPointer client_data;
```

ccc   Specifies the CCC.

compression_proc Specifies the gamut compression procedure that is to be applied when a color lies outside the screen's color gamut. If NULL and when functions using this CCC must convert a color specification to a device-dependent format and encounters a color that lies outside the screen's color gamut, that function will return *XcmsFailure*.

client_data   Specifies client data for the gamut compression procedure or NULL.

The *XcmsSetCompressionProc* function first sets the gamut compression procedure and client data in the specified CCC with the newly specified procedure and client data and then returns the old procedure.

To set the white point adjustment procedure and corresponding client data in a specified CCC, use *XcmsSetWhiteAdjustProc*.

```
XcmsWhiteAdjustProc XcmsSetWhiteAdjustProc(ccc, white_adjust_proc,
        client_data)
    XcmsCCC ccc;
    XcmsWhiteAdjustProc white_adjust_proc;
    XPointer client_data;
```

ccc   Specifies the CCC.

white_adjust_proc Specifies the white point adjustment procedure.

client_data   Specifies client data for the white point adjustment procedure or NULL.

The *XcmsSetWhiteAdjustProc* function first sets the white point adjustment procedure and client data in the specified CCC with the newly specified procedure and client data and then returns the old procedure.

7.8.5   Creating and Freeing a Color Conversion Context

You can explicitly create a CCC within your application by calling *XcmsCreateCCC*. These created CCCs can then be used by those functions that explicitly call for a CCC argument. Old CCCs that will not be used by the application should be freed using *XcmsFreeCCC*.

To create a CCC, use *XcmsCreateCCC*.

```
XcmsCCC XcmsCreateCCC(display, screen_number, visual,
        client_white_point, compression_proc, compression_client_data,
        white_adjust_proc, white_adjust_client_data)
    Display *display;
    int screen_number;
    Visual *visual;
    XcmsColor *client_white_point;
    XcmsCompressionProc compression_proc;
    XPointer compression_client_data;
    XcmsWhiteAdjustProc white_adjust_proc;
    XPointer white_adjust_client_data;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

visual   Specifies the visual type.

client_white_point Specifies the Client White Point. If NULL, the Client White Point is to be assumed to be the same as the Screen White Point. Note that the pixel member is ignored.

compression_proc Specifies the gamut compression procedure that is to be applied when a color lies outside the screen's color gamut. If NULL and when functions using this CCC must convert a color specification to a device-dependent format and encounters a color that lies outside the screen's color gamut, that function will return *XcmsFailure*.

compression_client_data
                        Specifies client data for use by the gamut compression procedure or NULL.

white_adjust_proc Specifies the white adjustment procedure that is to be applied when the Client White Point differs from the Screen White Point. NULL indicates that no white point adjustment is desired.

white_adjust_client_data
> Specifies client data for use with the white point adjustment procedure or NULL.

The *XcmsCreateCCC* function creates a CCC for the specified display, screen and visual.

To free a CCC, use *XcmsFreeCCC*.

```
void XcmsFreeCCC(ccc)
      XcmsCCC ccc;
```

ccc   Specifies the CCC.

The *XcmsFreeCCC* function frees the memory used for the specified CCC. Note that default CCCs and those currently associated with colormaps are ignored.

7.9   Converting Between Color Spaces

To convert an array of color specifications in arbitrary color formats to a single destination format, use *XcmsConvertColors*.

```
Status XcmsConvertColors(ccc, colors_in_out, ncolors,
        target_format, compression_flags_return)
    XcmsCCC ccc;
    XcmsColor colors_in_out[];
    unsigned int ncolors;
    XcmsColorFormat target_format;
    Bool compression_flags_return[];
```

ccc   Specifies the CCC. If conversion is between device-independent color spaces only (for example, TekHVC to CIELuv), the CCC is necessary only to specify the Client White Point.

colors_in_out   Specifies an array of color specifications. Pixel members are ignored and remain unchanged upon return.

ncolors   Specifies the number of *XcmsColor* structures in the color specification array.

target_format   Specifies the target color specification format.

compression_flags_return
                   Specifies an array of Boolean values for returning compression status. If a non-NULL pointer is supplied, each element of the array is set to *True* if the corresponding color was compressed, and *False* otherwise. Pass NULL if the compression status is not useful.

The *XcmsConvertColors* function converts the color specifications in the specified array of *XcmsColor* structures from their current format to a single target format, using the specified CCC. When the return value is *XcmsFailure*, the contents of the color specification array are left unchanged.

The array may contain a mixture of color specification formats (for example, 3 CIE XYZ, 2 CIE Luv, ...). Note that when the array contains both device-independent and device-dependent color specifications, and the target_format argument specifies a device-dependent format (for example, *XcmsRGBiFormat*, *XcmsRGBFormat*) all specifications are converted to CIE XYZ format then to the target device-dependent format.

7.10   Callback Functions

This section describes the gamut compression and white point adjustment callbacks.

The gamut compression procedure specified in the Color Conversion Context is called when an attempt to convert a color specification from *XcmsCIEXYZ* to a device-dependent format (typically *XcmsRGBi*) results in a color that lies outside the screen's color gamut. If the gamut compression procedure requires client data, this data is passed via the gamut compression client data in the CCC.

During color specification conversion between device-independent and device-dependent color spaces, if a white point adjustment procedure is specified in the CCC, it is triggered when the Client White Point and Screen White Point differ. If required, the client data is obtained from the CCC.

7.10.1   Prototype Gamut Compression Procedure

The gamut compression callback interface must adhere to the following:

```
Atypedef Status (*XcmsCompressionProc)(ccc, colors_in_out,\
        ncolors, index, compression_flags_return)
    XcmsCCC ccc;
    XcmsColor colors_in_out[];
    unsigned int ncolors;
    unsigned int index;
    Bool compression_flags_return[];
```

ccc   Specifies the CCC.

colors_in_out   Specifies an array of color specifications. Pixel members are ignored and remain unchanged upon return.

ncolors   Specifies the number of *XcmsColor* structures in the color specification array.

index   Specifies the index into the array of *XcmsColor* structures for the encountered color specification that lies outside the *Screen*'s color gamut. Valid values are 0 (for the first element) to ncolors −1.

compression_flags_return
                     Specifies an array of Boolean values for returning compression status. If a non-NULL pointer is supplied, and a color at a given index is compressed, then *True* should be stored at the corresponding index in this array.

When implementing a gamut compression procedure, consider the following rules and assumptions:

- The gamut compression procedure can attempt to compress one or multiple specifications at a time.

- When called, elements 0 to index −1 in the array of color specification array can be assumed to fall within the screen's color gamut. In addition these color specifications are already in some device-dependent format (typically *XcmsRGBi*). If any modifications are made to these color specifications, they must upon return be in their initial device-dependent format.

- When called, the element in the color specification array specified by the index argument contains the color specification outside the screen's color gamut encountered by the calling routine. In addition this color specification can be assumed to be in *XcmsCIEXYZ*. Upon return, this color specification must be in *XcmsCIEXYZ*.

- When called, elements from index to ncolors −1 in the color specification array may or may not fall within the screen's color gamut. In addition these color specifications can be assumed to be in *XcmsCIEXYZ*. If any modifications are made to these color specifications, they must upon return be in *XcmsCIEXYZ*.

- The color specifications passed to the gamut compression procedure have already been adjusted to the Screen White Point. This means that at this point the color specification's white point is the Screen White Point.

- If the gamut compression procedure uses a device-independent color space not initially accessible for use in the color management system, use *XcmsAddColorSpace* to insure that it is added.

7.10.2 Supplied Gamut Compression Procedures

The following equations are useful in describing gamut compression procedures.

$$CIELab\ Psychometric\ Chroma = sqrt\,(a\_star^2 + b\_star^2)$$

$$CIELab\ Psychometric\ Hue = \tan^{-1}\left[\frac{b\_star}{a\_star}\right]$$

$$CIELuv\ Psychometric\ Chroma = sqrt\,(u\_star^2 + v\_star^2)$$

$$CIELuv\ Psychometric\ Hue = \tan^{-1}\left[\frac{v\_star}{u\_star}\right]$$

The gamut compression callback procedures provided by Xlib are as follows.

XcmsCIELabClipL
Brings the encountered out of gamut color specification into the screen's color gamut by reducing or increasing CIE metric lightness (L*) in the CIE L*a*b* color space until the color is within the gamut. If the Psychometric Chroma of the color specification is beyond maximum for the Psychometric Hue Angle, then, while maintaining the same Psychometric Hue Angle, the color will be clipped to the CIE L*a*b* coordinates of maximum Psychometric Chroma. See *XcmsCIELabQueryMaxC*. No client data is necessary.

XcmsCIELabClipab
Brings the encountered out of gamut color specification into the screen's color gamut by reducing Psychometric Chroma, while maintaining Psychometric Hue Angle, until the color is within the gamut. No client data is necessary.

XcmsCIELabClipLab
Brings the encountered out of gamut color specification into the screen's color gamut by replacing it with CIE L*a*b* coordinates that fall within the color gamut while maintaining the original Psychometric Hue Angle and whose vector to the original coordinates is the shortest attainable. No client data is necessary.

XcmsCIELuvClipL
Brings the encountered out of gamut color specification into the screen's color gamut by reducing or increasing CIE metric lightness (L*) in the CIE L*u*v* color space until the color is within the gamut. If the Psychometric Chroma of the color specification is beyond maximum for the Psychometric Hue Angle, then, while maintaining the same Psychometric Hue Angle, the color will be clipped to the CIE L*u*v* coordinates of maximum Psychometric Chroma. See *XcmsCIELuvQueryMaxC*. No client data is necessary.

XcmsCIELuvClipuv
> Brings the encountered out of gamut color specification into the screen's color gamut by reducing Psychometric Chroma while maintaining Psychometric Hue Angle, until the color is within the gamut. No client data is necessary.

XcmsCIELuvClipLuv
> Brings the encountered out of gamut color specification into the screen's color gamut by replacing it with CIE L*u*v* coordinates that fall within the color gamut while maintaining the original Psychometric Hue Angle and whose vector to the original coordinates is the shortest attainable. No client data is necessary.

XcmsTekHVCClipV
> Brings the encountered out of gamut color specification into the screen's color gamut by reducing or increasing the Value dimension in the TekHVC color space until the color is within the gamut. If Chroma of the color specification is beyond maximum for the particular Hue, then, while maintaining the same Hue, the color will be clipped to the Value and Chroma coordinates that represent maximum Chroma for that particular Hue. No client data is necessary.

XcmsTekHVCClipC
> Brings the encountered out of gamut color specification into the screen's color gamut by reducing the Chroma dimension in the TekHVC color space until the color is within the gamut. No client data is necessary.

XcmsTekHVCClipVC
> Brings the encountered out of gamut color specification into the screen's color gamut by replacing it with TekHVC coordinates that fall within the color gamut while maintaining the original Hue and whose vector to the original coordinates is the shortest attainable. No client data is necessary.

7.10.3 Prototype White Point Adjustment Procedure

The white point adjustment procedure interface must adhere to the following:

```
typedef Status (*XcmsWhiteAdjustProc)(ccc, initial_white_point,
        target_white_point, target_format, colors_in_out,
        ncolors, compression_flags_return)
    XcmsCCC ccc;
    XcmsColor *initial_white_point;
    XcmsColor *target_white_point;
    XcmsColorFormat target_format;
    XcmsColor colors_in_out[];
    unsigned int ncolors;
    Bool compression_flags_return[];
```

ccc   Specifies the CCC.

initial_white_point Specifies the initial white point.

target_white_point Specifies the target white point.

target_format   Specifies the target color specification format.

colors_in_out   Specifies an array of color specifications. Pixel members are ignored and remain unchanged upon return.

ncolors   Specifies the number of *XcmsColor* structures in the color specification array.

compression_flags_return

Specifies an array of Boolean values for returning compression status. If a non-NULL pointer is supplied, and a color at a given index is compressed, then *True* should be stored at the corresponding index in this array.

7.10.4 Supplied White Point Adjustment Procedures

White point adjustment procedures provided by Xlib are as follows.

XcmsCIELabWhiteShiftColors

Uses the CIE L*a*b* color space for adjusting the chromatic character of colors to compensate for the chromatic differences between the source and destination white points. This procedure simply converts the color specifications to *XcmsCIELab* using the source white point and then converts to the target specification format using the destinations white point. No client data is necessary.

XcmsCIELuvWhiteShiftColors

Uses the CIE L*u*v* color space for adjusting the chromatic character of colors to compensate for the chromatic differences between the source and destination white points. This procedure simply converts the color specifications to *XcmsCIELuv* using the source white point and then converts to the target specification format using the destinations white point. No client data is necessary.

XcmsTekHVCWhiteShiftColors

Uses the TekHVC color space for adjusting the chromatic character of colors to compensate for the chromatic differences between the source and destination white points. This procedure simply converts the color specifications to *XcmsTekHVC* using the source white point and then converts to the target specification format using the destinations white point. An advantage of this procedure over those previously described is an attempt to minimize hue shift. No client data is necessary.

From an implementation point of view, these white point adjustment procedures convert the color specifications to a device-independent but white-point-dependent color space (for example, CIE L*u*v*, CIE L*a*b*, TekHVC) using one white point and then converting those specifications to the target color space using another white point. In other words, the specification goes in the color space with one white point but comes out with another white point, resulting in a chromatic shift based on the chromatic displacement between the initial white point and target white point. The CIE color spaces that are assumed to be white-point-independent are CIE u'v'Y, CIE XYZ, and CIE xyY. When developing a custom white point adjustment procedure that uses a device-independent color space not initially accessible for use in the color management system, use *XcmsAddColorSpace* to insure that it is added.

As an example, if a white point adjustment procedure is specified by the Color Conversion Context and if the Client White Point and Screen White Point differ, the *XcmsAllocColor* function will use the white point adjustment procedure twice:

- once to convert to *XcmsRGB*
- a second time to convert from *XcmsRGB*.

For example, assume the specification is in *XcmsCIEuvY* and the adjustment procedure is *XcmsCIELuvWhiteShiftColors*. During conversion to *XcmsRGB*, the call to *XcmsAllocColor* results in the following series of color specification conversions:

- from *XcmsCIEuvY* to *XcmsCIELuv* using the Client White Point
- from *XcmsCIELuv* to *XcmsCIEuvY* using the Screen White Point

- from *XcmsCIEuvY* to *XcmsCIEXYZ* (CIE u'v'Y and XYZ are white-point-independent color spaces)

- from *XcmsCIEXYZ* to *XcmsRGBi*

- finally, from *XcmsRGBi* to *XcmsRGB.*

The resulting RGB specification is passed to *XAllocColor* and the RGB specification returned by *XAllocColor* is converted back to *XcmsCIEuvY* by reversing the color conversion sequence.

7.11   Gamut Querying Functions

This section describes the gamut querying functions that are provided by Xlib. These functions allow the client to query the boundary of the screen's color gamut in terms of the CIE L*a*b*, CIE L*u*v*, and TekHVC color spaces. Functions are also provided that allow you to query the color specification of:

- White (full intensity red, green and blue)

- Red (full intensity red while green and blue are zero)

- Green (full intensity green while red and blue are zero)

- Blue (full intensity blue while red and green are zero)

- Black (zero intensity red, green and blue).

The white point associated with color specifications passed to and returned from these gamut querying functions are assumed to be the Screen White Point. This is a reasonable assumption, since the client is trying to query the screen's color gamut.

Note that the following naming convention is used for the ''Max'' and ''Min'' functions:

```
Xcms<color_space>QueryMax<dimensions>

Xcms<color_space>QueryMin<dimensions>
```

Note that the *<dimensions>* consists of letter or letters that identify the dimension or dimensions of the color space that are not fixed. For example, *XcmsTekHVCQueryMaxC* is given a fixed Hue and Value for which maximum Chroma is found.

7.11.1   Red, Green and Blue Queries

To obtain the color specification for black (zero intensity red, green and blue), use *XcmsQueryBlack*.

```
Status XcmsQueryBlack(ccc, target_format, color_return)
      XcmsCCC ccc;
      XcmsColorFormat target_format;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

target_format   Specifies the target color specification format.

color_return   Returns the color specification in the specified target format for zero intensity red, green and blue. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsQueryBlack* function returns the color specification in the specified target format for zero intensity red, green and blue.

To obtain the color specification for blue (full intensity blue while red and green are zero), use *XcmsQueryBlue.*

```
Status XcmsQueryBlue(ccc, target_format, color_return)
      XcmsCCC ccc;
      XcmsColorFormat target_format;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                     Adjustment procedures are ignored.

target_format   Specifies the target color specification format.

color_return   Returns the color specification in the specified target format for full intensity
                     blue while red and green are zero. The white point associated with the
                     returned color specification is the Screen White Point. The value returned in
                     the pixel member is undefined.

The *XcmsQueryBlue* function returns the color specification in the specified target format for full
intensity blue while red and green are zero.

To obtain the color specification for green (full intensity green while red and blue are zero), use
XcmsQueryGreen.

```
Status XcmsQueryGreen(ccc, target_format, color_return)
      XcmsCCC ccc;
      XcmsColorFormat target_format;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                     Adjustment procedures are ignored.

target_format   Specifies the target color specification format.

color_return   Returns the color specification in the specified target format for full intensity
                     green while red and blue are zero. The white point associated with the
                     returned color specification is the Screen White Point. The value returned in
                     the pixel member is undefined.

The *XcmsQueryGreen* function returns the color specification in the specified target format for
full intensity green while red and blue are zero.

To obtain the color specification for red (full intensity red while green and blue are zero), use
XcmsQueryRed.

```
Status XcmsQueryRed(ccc, target_format, color_return)
      XcmsCCC ccc;
      XcmsColorFormat target_format;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                     Adjustment procedures are ignored.

target_format   Specifies the target color specification format.

color_return   Returns the color specification in the specified target format for full intensity
                     red while green and blue are zero. The white point associated with the
                     returned color specification is the Screen White Point. The value returned in
                     the pixel member is undefined.

The *XcmsQueryRed* function returns the color specification in the specified target format for full
intensity red while green and blue are zero.

To obtain the color specification for white (full intensity red, green and blue), use *XcmsQueryWhite*.

```
Status XcmsQueryWhite(ccc, target_format, color_return)
      XcmsCCC ccc;
      XcmsColorFormat target_format;
      XcmsColor *color_return;
```

ccc　　　　　　Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

target_format　Specifies the target color specification format.

color_return　Returns the color specification in the specified target format for full intensity red, green and blue. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsQueryWhite* function returns the color specification in the specified target format for full intensity red, green and blue.

7.11.2 CIELab Queries

To obtain the CIE L*a*b* coordinates of maximum Psychometric Chroma for a given Psychometric Hue Angle and CIE metric lightness (L*), use *XcmsCIELabQueryMaxC*.

CIELab Psychometric Chroma $= sqrt(a\_star^2 + b\_star^2)$

CIELab Psychometric Hue $= \tan^{-1}\left[\dfrac{b\_star}{a\_star}\right]$

```
Status XcmsCIELabQueryMaxC(ccc, hue_angle, L_star, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsFloat L_star;
      XcmsColor *color_return;
```

ccc　　　　　　Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle　　Specifies the hue angle in degrees at which to find maximum chroma.

L_star　　　　Specifies the lightness (L*) at which to find maximum chroma.

color_return　Returns the CIE L*a*b* coordinates of maximum chroma displayable by the screen for the given hue angle and lightness. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELabQueryMaxC* function, given a hue angle and lightness, finds the point of maximum chroma displayable by the screen. It returns this point in CIE L*a*b* coordinates.

To obtain the CIE L*a*b* coordinates of maximum CIE metric lightness (L*) for a given Psychometric Hue Angle and Psychometric Chroma, use *XcmsCIELabQueryMaxL*.

```
Status XcmsCIELabQueryMaxL(ccc, hue_angle, chroma, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsFloat chroma;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle   Specifies the hue angle in degrees at which to find maximum lightness.

chroma   Specifies the chroma at which to find maximum lightness.

color_return   Returns the CIE L*a*b* coordinates of maximum lightness displayable by the screen for the given hue angle and chroma. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELabQueryMaxL* function, given a hue angle and chroma, finds the point in CIE L*a*b* color space of maximum lightness (L*) displayable by the screen. It returns this point in CIE L*a*b* coordinates. An *XcmsFailure* return value usually indicates that the given chroma is beyond maximum for the given hue angle.

To obtain the CIE L*a*b* coordinates of maximum Psychometric Chroma for a given Psychometric Hue Angle, use *XcmsCIELabQueryMaxLC*.

```
Status XcmsCIELabQueryMaxLC(ccc, hue_angle, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle   Specifies the hue angle in degrees at which to find maximum chroma.

color_return   Returns the CIE L*a*b* coordinates of maximum chroma displayable by the screen for the given hue angle. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELabQueryMaxLC* function, given a hue angle, finds the point of maximum chroma displayable by the screen. It returns this point in CIE L*a*b* coordinates.

To obtain the CIE L*a*b* coordinates of minimum CIE metric lightness (L*) for a given Psychometric Hue Angle and Psychometric Chroma, use *XcmsCIELabQueryMinL*.

```
Status XcmsCIELabQueryMinL(ccc, hue_angle, chroma, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsFloat chroma;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle   Specifies the hue angle in degrees at which to find minimum lightness.

chroma   Specifies the chroma at which to find minimum lightness.

    color_return   Returns the CIE L*a*b* coordinates of minimum lightness displayable by the screen for the given hue angle and chroma. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELabQueryMinL* function, given a hue angle and chroma, finds the point of minimum lightness (L*) displayable by the screen. It returns this point in CIE L*a*b* coordinates. An *XcmsFailure* return value usually indicates that the given chroma is beyond maximum for the given hue angle.

7.11.3 CIELuv Queries

To obtain the CIE L*u*v* coordinates of maximum Psychometric Chroma for a given Psychometric Hue Angle and CIE metric lightness (L*), use *XcmsCIELuvQueryMaxC*.

$$CIELuv\ Psychometric\ Chroma = sqrt\,(u\_star^2 + v\_star^2)$$

$$CIELuv\ Psychometric\ Hue = \tan^{-1}\left[\frac{v\_star}{u\_star}\right]$$

```
Status XcmsCIELuvQueryMaxC(ccc, hue_angle, L_star, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsFloat L_star;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle   Specifies the hue angle in degrees at which to find maximum chroma.

L_star   Specifies the lightness (L*) at which to find maximum chroma.

color_return   Returns the CIE L*u*v* coordinates of maximum chroma displayable by the screen for the given hue angle and lightness. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELuvQueryMaxC* function, given a hue angle and lightness, finds the point of maximum chroma displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates.

To obtain the CIE L*u*v* coordinates of maximum CIE metric lightness (L*) for a given Psychometric Hue Angle and Psychometric Chroma, use *XcmsCIELuvQueryMaxL*.

```
Status XcmsCIELuvQueryMaxL(ccc, hue_angle, chroma, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsFloat chroma;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle   Specifies the hue angle in degrees at which to find maximum lightness.

L_star   Specifies the lightness (L*) at which to find maximum lightness.

color_return   Returns the CIE L*u*v* coordinates of maximum lightness displayable by the screen for the given hue angle and chroma. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELuvQueryMaxL* function, given a hue angle and chroma, finds the point in CIE L*u*v* color space of maximum lightness (L*) displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates. An *XcmsFailure* return value usually indicates that the given chroma is beyond maximum for the given hue angle.

To obtain the CIE L*u*v* coordinates of maximum Psychometric Chroma for a given Psychometric Hue Angle, use *XcmsCIELuvQueryMaxLC*.

```
Status XcmsCIELuvQueryMaxLC(ccc, hue_angle, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle   Specifies the hue angle in degrees at which to find maximum chroma.

color_return   Returns the CIE L*u*v* coordinates of maximum chroma displayable by the screen for the given hue angle. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELuvQueryMaxLC* function, given a hue angle, finds the point of maximum chroma displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates.

To obtain the CIE L*u*v* coordinates of minimum CIE metric lightness (L*) for a given Psychometric Hue Angle and Psychometric Chroma, use *XcmsCIELuvQueryMinL*.

```
Status XcmsCIELuvQueryMinL(ccc, hue_angle, chroma, color_return)
      XcmsCCC ccc;
      XcmsFloat hue_angle;
      XcmsFloat chroma;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

hue_angle   Specifies the hue angle in degrees at which to find minimum lightness.

chroma   Specifies the chroma at which to find minimum lightness.

color_return   Returns the CIE L*u*v* coordinates of minimum lightness displayable by the screen for the given hue angle and chroma. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

The *XcmsCIELuvQueryMinL* function, given a hue angle and chroma, finds the point of minimum lightness (L*) displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates. An *XcmsFailure* return value usually indicates that the given chroma is beyond maximum for the given hue angle.

7.11.4   TekHVC Queries

To obtain the maximum Chroma for a given Hue and Value, use *XcmsTekHVCQueryMaxC*.

```
Status XcmsTekHVCQueryMaxC(ccc, hue, value, color_return)
      XcmsCCC ccc;
      XcmsFloat hue;
      XcmsFloat value;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                   Adjustment procedures are ignored.

hue   Specifies the Hue in which to find the maximum Chroma.

value   Specifies the Value in which to find the maximum Chroma.

color_return   Returns the maximum Chroma along with the actual Hue and Value at which
                   the maximum Chroma was found. The white point associated with the
                   returned color specification is the Screen White Point. The value returned in
                   the pixel member is undefined.

The *XcmsTekHVCQueryMaxC* function, given a Hue and Value, determines the maximum
Chroma in TekHVC color space displayable by the screen. Note that it returns the maximum
Chroma along with the actual Hue and Value at which the maximum Chroma was found.

To obtain the maximum Value for a given Hue and Chroma, use *XcmsTekHVCQueryMaxV*.

```
Status XcmsTekHVCQueryMaxV(ccc, hue, chroma, color_return)
      XcmsCCC ccc;
      XcmsFloat hue;
      XcmsFloat chroma;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                   Adjustment procedures are ignored.

hue   Specifies the Hue in which to find the maximum Value.

chroma   Specifies the chroma at which to find maximum Value.

color_return   Returns the maximum Value along with the Hue and Chroma at which the
                   maximum Value was found. The white point associated with the returned
                   color specification is the Screen White Point. The value returned in the pixel
                   member is undefined.

The *XcmsTekHVCQueryMaxV* function, given a Hue and Chroma, determines the maximum
Value in TekHVC color space displayable by the screen. Note that it returns the maximum Value
and the actual Hue and Chroma at which the maximum Value was found.

To obtain the maximum Chroma and Value at which it is reached for a specified Hue, use
XcmsTekHVCQueryMaxVC.

```
Status XcmsTekHVCQueryMaxVC(ccc, hue, color_return)
      XcmsCCC ccc;
      XcmsFloat hue;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                   Adjustment procedures are ignored.

hue   Specifies the Hue in which to find the maximum Chroma.

color_return   Returns the color specification in XcmsTekHVC for the maximum Chroma,
                         the Value at which that maximum Chroma is reached and actual Hue at which
                         the maximum Chroma was found. The white point associated with the
                         returned color specification is the Screen White Point. The value returned in
                         the pixel member is undefined.

The *XcmsTekHVCQueryMaxVC* function, given a Hue, determines the maximum Chroma in
TekHVC color space displayable by the screen and the Value at which that maximum Chroma is
reached. Note that it returns the maximum Chroma, the Value at which that maximum Chroma
is reached, and the actual Hue for which the maximum Chroma was found.

To obtain a specified number of TekHVC specifications such that they contain a maximum
Values for a specified Hue, and the Chroma at which the maximum Values are reached, use
XcmsTekHVCQueryMaxVSamples.

```
Status XcmsTekHVCQueryMaxVSamples(ccc, hue, colors_return, nsamples)
      XcmsCCC ccc;
      XcmsFloat hue;
      XcmsColor colors_return[];
      unsigned int nsamples;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                         Adjustment procedures are ignored.

hue   Specifies the Hue for maximum Chroma/Value samples.

nsamples   Specifies the number of samples.

colors_in_out   Returns nsamples of color specifications in XcmsTekHVC such that the
                         Chroma is the maximum attainable for the Value and Hue. The white point
                         associated with the returned color specification is the Screen White Point. The
                         value returned in the pixel member is undefined.

The *XcmsTekHVCQueryMaxVSamples* returns nsamples of maximum Value, Chroma at which
that maximum Value is reached, and the actual Hue for which the maximum Chroma was
found. These sample points may then be used to plot the maximum Value/Chroma boundary of
the screen's color gamut for the specified Hue in TekHVC color space.

To obtain the minimum Value for a given Hue and Chroma, use *XcmsTekHVCQueryMinV*.

```
Status XcmsTekHVCQueryMinV(ccc, hue, chroma, color_return)
      XcmsCCC ccc;
      XcmsFloat hue;
      XcmsFloat chroma;
      XcmsColor *color_return;
```

ccc   Specifies the CCC. Note that the CCC's Client White Point and White Point
                         Adjustment procedures are ignored.

hue   Specifies the Hue in which to find the minimum Value.

value   Specifies the Value in which to find the minimum Value.

color_return   Returns the minimum Value and the actual Hue and Chroma at which the
                         minimum Value was found. The white point associated with the returned
                         color specification is the Screen White Point. The value returned in the pixel
                         member is undefined.

The *XcmsTekHVCQueryMinV* function, given a Hue and Chroma, determines the minimum Value in TekHVC color space displayable by the screen. Note that it returns the minimum Value and the actual Hue and Chroma at which the minimum Value was found.

7.12 Color Management Extensions

The Xlib color management facilities can be extended in two ways:

- Device-independent Color Spaces

- device-independent color spaces that are derivable to CIE XYZ space can be added using the *XcmsAddColorSpace* function

- Color Characterization Function Set

- a Color Characterization Function Set consists of device-dependent color spaces and their functions that convert between these color spaces and the CIE XYZ color space, bundled together for a specific class of output devices. A function set can be added using the *XcmsAddFunctionSet* function.

7.12.1 Color Spaces

The CIE XYZ color space serves as the ''hub'' for all conversions between device-independent and device-dependent color spaces. Therefore, associated with each color space is the knowledge to convert an *XcmsColor* structure to and from CIE XYZ format. For example, conversion from CIE L*u*v* to RGB requires the knowledge to convert from CIE L*u*v* to CIE XYZ and from CIE XYZ to RGB. This knowledge is stored as an array of functions that when applied in series will convert the *XcmsColor* structure to or from CIE XYZ format. This color specification conversion mechanism facilitates the addition of color spaces.

Of course, when converting between only device-independent color spaces or only device-dependent color spaces, short cuts are taken whenever possible. For example, conversion from TekHVC to CIE L*u*v* is performed by intermediate conversion to CIE u*v*Y and then to CIE L*u*v*, thus bypassing conversion between CIE u*v*Y and CIE XYZ.

7.12.2 Adding Device-independent Color Spaces

To add a device-independent color space, use *XcmsAddColorSpace*.

```
Status XcmsAddColorSpace(color_space)
      XcmsColorSpace *color_space;
```

color_space   Specifies the device-independent color space to add.

The *XcmsAddColorSpace* function makes a device-independent color space (actually an *XcmsColorSpace* structure) accessible by the color management system. Because format values for unregistered color spaces are assigned at run-time, they should be treated as private to the client. If references to an unregistered color space must be made outside the client (for example, storing color specifications in a file using the unregistered color space), then reference should be made by color space prefix (see *XcmsFormatOfPrefix* and *XcmsPrefixOfFormat*).

If the *XcmsColorSpace* structure is already accessible in the color management system, *XcmsAddColorSpace* returns *XcmsSuccess*.

Note that added *XcmsColorSpaces* must be retained for reference by Xlib.

7.12.3   Querying Color Space Format and Prefix

To obtain the format associated with the color space associated with a specified color string prefix, use *XcmsFormatOfPrefix*.

```
XcmsColorFormat XcmsFormatOfPrefix(prefix)
      char *prefix;
```

prefix   Specifies the string that contains the color space prefix.

The *XcmsFormatOfPrefix* function returns format for the specified color space prefix (for example, ''CIEXYZ''). Note that the prefix is case-insensitive. If the color space is not accessible in the color management system, *XcmsFormatOfPrefix* returns *XcmsUndefinedFormat*.

To obtain the color string prefix associated with the color space specified by a color format, use *XcmsPrefixOfFormat*.

```
char *XcmsPrefixOfFormat(format)
      XcmsColorFormat format;
```

format   Specifies the color specification format.

The *XcmsPrefixOfFormat* function returns the string prefix associated with the color specification encoding specified by format. Otherwise, if none is found, it returns NULL. Note that the returned string must be treated as read-only.

7.12.4   Creating Additional Color Spaces

Color space-specific information necessary for color space conversion and color string parsing is stored in an *XcmsColorSpace* structure. Therefore, a new structure containing this information is required for each additional color space. In the case of device-independent color spaces, a handle to this new structure (that is, by means of a global variable) is usually made accessible to the client program for use with the *XcmsAddColorSpace* function.

If a new *XcmsColorSpace* structure specifies a color space not registered with the X Consortium, because format values for unregistered color spaces are assigned at run-time they should be treated as private to the client. If references to an unregistered color space must be made outside the client (for example, storing color specifications in a file using the unregistered color space), then reference should be made by color space prefix (see *XcmsFormatOfPrefix* and *XcmsPrefixOfFormat*).

```
typedef (*XcmsConversionProc)();
typedef XcmsConversionProc *XcmsFuncListPtr;
          /* A NULL terminated list of function pointers*/


typedef struct _XcmsColorSpace {
      char *prefix;
      XcmsColorFormat format;
      XcmsParseStringProc parseString;
      XcmsFuncListPtr to_CIEXYZ;
      XcmsFuncListPtr from_CIEXYZ;
      int inverse_flag;
} XcmsColorSpace;
```

The prefix member specifies the prefix that indicates a color string is in this color space's string format. For example, ''ciexyz'' or ''CIEXYZ'' for CIE XYZ, and ''rgb'' or ''RGB'' for RGB. Note that the prefix is case-insensitive. The format member specifies the color specification format. Formats for unregistered color spaces are assigned at run-time. The parseString member

contains a pointer to the function that can parse a color string into an *XcmsColor* structure. This function returns an integer (int): non-zero if it succeeded and zero otherwise. The to_CIEXYZ and from_CIEXYZ members contain pointers, each to a NULL terminated list of function pointers. When the list of functions are executed in series, it will convert the color specified in an *XcmsColor* structure from/to the current color space format to/from the CIE XYZ format. Each function returns an integer (int): non-zero if it succeeded and zero otherwise. Note that the white point to be associated with the colors is specified explicitly, even though white points can be found in the Color Conversion Context. The inverse_flag member, if non-zero, specifies that for each function listed in to_CIEXYZ, its inverse function can be found in from_CIEXYZ such that:

```
Given: n = number of functions in each list

foreach i, such that 0 <= i < n
    from_CIEXYZ[n - i - 1] is the inverse of to_CIEXYZ[i].
```

This allows Xlib to use the shortest conversion path, thus, bypassing CIE XYZ if possible (for example, TekHVC to CIE L*u*v*).

7.12.5   Parse String Callback

The callback in the *XcmsColorSpace* structure for parsing a color string for the particular color space must adhere to the following software interface specification:

```
typedef int (*XcmsParseStringProc)(color_string, color_return)
    char *color_string;
    XcmsColor *color_return;   /* color to compress */
```

color_string   Specifies the color string to parse.

color_return   Returns the color specification in the color space's format.

7.12.6   Color Specification Conversion Callback

Callback functions in the *XcmsColorSpace* structure for converting a color specification between device-independent spaces must adhere to the following software interface specification:

```
Status ConversionProc(ccc, white_point, colors_in_out, ncolors)
    XcmsCCC ccc;
    XcmsColor *white_point;
    XcmsColor *colors_in_out;
    unsigned int ncolors;
```

ccc   Specifies the CCC.

white_point   Specifies the white point associated with color specifications. Note that the pixel member is ignored and that the color specification is left unchanged upon return.

colors_in_out   Specifies an array of color specifications. Pixel members are ignored and remain unchanged upon return.

ncolors   Specifies the number of *XcmsColor* structures in the color specification array.

Callback functions in the *XcmsColorSpace* structure for converting a color specification to or from a device-dependent space must adhere to the following software interface specification:

```
Status ConversionProc(ccc, colors_in_out, ncolors, compression_flags_return)
      XcmsCCC ccc;
      XcmsColor *colors_in_out;
      unsigned int ncolors;
      Bool compression_flags_return[];
```

ccc   Specifies the CCC.

colors_in_out   Specifies an array of color specifications. Pixel members are ignored and
                      remain unchanged upon return.

ncolors   Specifies the number of *XcmsColor* structures in the color specification array.

compression_flags_return
                      Specifies an array of Boolean values for returning compression status. If a
                      non-NULL pointer is supplied, and a color at a given index is compressed,
                      then *True* should be stored at the corresponding index in this array.

Conversion functions are available globally for use by other color spaces. The conversion
functions provided by Xlib are:

Function	Converts
XcmsCIELabToCIEXYZ	From XcmsCIELabFormat to XcmsCIEXYZFormat
XcmsCIELuvToCIEuvY	From XcmsCIELuvFormat to XcmsCIEuvYFormat
XcmsCIEXYZToCIELab	From XcmsCIEXYZFormat to XcmsCIELabFormat
XcmsCIEXYZToCIEuvY	From XcmsCIEXYZFormat to XcmsCIEuvYFormat
XcmsCIEXYZToCIExyY	From XcmsCIEXYZFormat to XcmsCIExyYFormat
XcmsCIEXYZToRGBi	From XcmsCIEXYZFormat to XcmsRGBiFormat
XcmsCIEuvYToCIELuv	From XcmsCIEuvYFormat to XcmsCIELabFormat
XcmsCIEuvYToCIEXYZ	From XcmsCIEuvYFormat to XcmsCIEXYZFormat
XcmsCIEuvYToTekHVC	From XcmsCIEuvYFormat to XcmsTekHVCFormat
XcmsCIExyYToCIEXYZ	From XcmsCIExyYFormat to XcmsCIEXYZFormat
XcmsRGBToRGBi	From XcmsRGBFormat to XcmsRGBiFormat
XcmsRGBiToCIEXYZ	From XcmsRGBiFormat to XcmsCIEXYZFormat
XcmsRGBiToRGB	From XcmsRGBiFormat to XcmsRGBFormat
XcmsTekHVCToCIEuvY	From XcmsTekHVCFormat to XcmsCIEuvYFormat

7.12.7 Function Sets

Functions to convert between device-dependent color spaces and CIE XYZ may differ for
different classes of output devices (for example, color versus gray monitors). Therefore, the
notion of a Color Characterization Function Set (hereafter referred to as a Function Set) has been
developed. A function set consists of device-dependent color spaces and the functions that
convert color specifications between these device-dependent color spaces and the CIE XYZ color
space appropriate for a particular class of output devices. The function set also contains a
function that reads color characterization data off root window properties. It is this
characterization data that will differ between devices within a class of output devices. For
details about color characterization data is stored in root window properties, see Device Color
Characterization in the **ICCCM** specification. The LINEAR_RGB Function Set is provided by
Xlib and will support most color monitors. Function sets may require data that differs from
those needed for the LINEAR_RGB Function Set. In that case, its corresponding data may be
stored on different root window properties.

7.12.8   Adding Function Sets

To add a Color Characterization Function Set, use *XcmsAddFunctionSet*.

```
Status XcmsAddFunctionSet(function_set)
      XcmsFunctionSet *function_set;
```

function_set   Specifies the Color Characterization Function Set to add.

The *XcmsAddFunctionSet* adds a Color Characterization Function Set to the color management system. If the Function Set uses device-dependent *XcmsColorSpace* structures not accessible in the color management system, *XcmsAddFunctionSet* adds them. If an added *XcmsColorSpace* structure is for a device-dependent color space not registered with the X Consortium, because format values for unregistered color spaces are assigned at run-time they should be treated as private to the client. If references to an unregistered color space must be made outside the client (for example, storing color specifications in a file using the unregistered color space), then reference should be made by color space prefix (see *XcmsFormatOfPrefix* and *XcmsPrefixOfFormat*).

Additional function sets should be added before any calls to other Xlib routines are made. If not, the *XcmsPerScrnInfo* member of a previously created *XcmsCCC* does not have the opportunity to initialize with the added Function Set.

7.12.9   Creating Additional Function Sets

Creation of additional Color Characterization Function Sets should be required only when an output device does not conform to existing function sets or when additional device-dependent color spaces are necessary. A function set consists primarily of a collection of device-dependent *XcmsColorSpace* structures and a means to read and store a screen's color characterization data. This data is stored in an *XcmsFunctionSet* structure. A handle to this structure (that is, by means of global variable) is usually made accessible to the client program for use with *XcmsAddFunctionSet*.

If a Function Set uses new device-dependent *XcmsColorSpace* structures, they will be transparently processed into the color management system. Function Sets can share an *XcmsColorSpace* structure for a device-dependent color space. In addition, multiple *XcmsColorSpace* structures are allowed for a device-dependent color space; however, a Function Set can reference only one of them. These *XcmsColorSpace* structures will differ in the functions to convert to and from CIE XYZ, thus tailored for the specific Function Set.

```
typedef struct _XcmsFunctionSet {
      XcmsColorSpace **DDColorSpaces;
      XcmsScreenInitProc screenInitProc;
      XcmsScreenFreeProc screenFreeProc;
} XcmsFunctionSet;
```

The DDColorSpaces member is a pointer to a NULL terminated list of pointers to *XcmsColorSpace* structures for the device-dependent color spaces that are supported by the Function Set. The screenInitProc member is set to the callback procedure (see following interface specification) that initializes the *XcmsPerScrnInfo* structure for a particular screen.

The screen initialization callback must adhere to the following software interface specification:

```
typedef Status (*XcmsScreenInitProc)(display, screen_number,
        screen_info)
    Display *display;
    int screen_number;
    XcmsPerScrnInfo *screen_info;
```

display   Specifies the connection to the X server.

screen_number   Specifies the appropriate screen number on the host server.

screen_info   Specifies the *XcmsPerScrnInfo* structure, which contains the per screen
                   information.

The screen initialization callback in the *XcmsFunctionSet* structure fetches the Color
Characterization Data (device profile) for the specified screen, typically off properties on the
screen's root window; then it initializes the specified *XcmsPerScrnInfo* structure. If successful, the
procedure fills in the *XcmsPerScrnInfo* structure as follows:

- It sets the screenData member to the address of the created device profile data structure
  (contents known only by the function set).

- It next sets the screenWhitePoint member.

- It next sets the functionSet member to the address of the *XcmsFunctionSet* structure.

- It then sets the state member to *XcmsInitSuccess* and finally returns *XcmsSuccess*.

If unsuccessful, the procedure sets the state member to *XcmsInitFailure* and returns *XcmsFailure*.

The *XcmsPerScrnInfo* structure contains:

```
typedef struct _XcmsPerScrnInfo {
    XcmsColor screenWhitePoint;
    XPointer functionSet;
    XPointer screenData;
    unsigned char state;
    char pad[3];
} XcmsPerScrnInfo;
```

The screenWhitePoint member specifies the white point inherent to the screen. The functionSet
member specifies the appropriate Function Set. The screenData member specifies the device
profile. The state member is set to one of the following:

- *XcmsInitNone* indicates initialization has not been previously attempted.

- *XcmsInitFailure* indicates initialization has been previously attempted but failed.

- *XcmsInitSuccess* indicates initialization has been previously attempted and succeeded.

The screen free callback must adhere to the following software interface specification:

```
typedef void (*XcmsScreenFreeProc)(screenData)
    XPointer screenData;
```

screenData   Specifies the data to be freed.

This function is called to free the screenData stored in an *XcmsPerScrnInfo* structure.

Graphics Context Functions

A number of resources are used when performing graphics operations in X. Most information about performing graphics (for example, foreground color, background color, line style, and so on) are stored in resources called graphics contexts (GC). Most graphics operations (see Chapter 9) take a GC as an argument. Although in theory the X protocol permits sharing of GCs between applications, it is expected that applications will use their own GCs when performing operations. Sharing of GCs is highly discouraged because the library may cache GC state.

Graphics operations can be performed to either windows or pixmaps, which collectively are called drawables. Each drawable exists on a single screen. A GC is created for a specific screen and drawable depth, and can only be used with drawables of matching screen and depth.

8.1   Manipulating Graphics Context/State

Most attributes of graphics operations are stored in Graphic Contexts (GCs). These include line width, line style, plane mask, foreground, background, tile, stipple, clipping region, end style, join style, and so on. Graphics operations (for example, drawing lines) use these values to determine the actual drawing operation. Extensions to X may add additional components to GCs. The contents of a GC are private to Xlib.

Xlib implements a write-back cache for all elements of a GC that are not resource IDs to allow Xlib to implement the transparent coalescing of changes to GCs. For example, a call to *XSetForeground* of a GC followed by a call to *XSetLineAttributes* results in only a single-change GC protocol request to the server. GCs are neither expected nor encouraged to be shared between client applications, so this write-back caching should present no problems. Applications cannot share GCs without external synchronization. Therefore, sharing GCs between applications is highly discouraged.

To set an attribute of a GC, set the appropriate member of the *XGCValues* structure and OR in the corresponding value bitmask in your subsequent calls to *XCreateGC*. The symbols for the value mask bits and the *XGCValues* structure are:

```
/* GC attribute value mask bits */


    #define    GCFunction            (1L<<0)
    #define    GCPlaneMask           (1L<<1)
    #define    GCForeground          (1L<<2)
    #define    GCBackground          (1L<<3)
    #define    GCLineWidth           (1L<<4)
    #define    GCLineStyle           (1L<<5)
    #define    GCCapStyle            (1L<<6)
    #define    GCJoinStyle           (1L<<7)
    #define    GCFillStyle           (1L<<8)
    #define    GCFillRule            (1L<<9)
    #define    GCTile                (1L<<10)
    #define    GCStipple             (1L<<11)
    #define    GCTileStipXOrigin     (1L<<12)
    #define    GCTileStipYOrigin     (1L<<13)
    #define    GCFont                (1L<<14)
    #define    GCSubwindowMode       (1L<<15)
    #define    GCGraphicsExposures   (1L<<16)
    #define    GCClipXOrigin         (1L<<17)
    #define    GCClipYOrigin         (1L<<18)
    #define    GCClipMask            (1L<<19)
    #define    GCDashOffset          (1L<<20)
    #define    GCDashList            (1L<<21)
    #define    GCArcMode             (1L<<22)


/* Values */

typedef struct {
    int function;              /* logical operation */
    unsigned long plane_mask; /* plane mask */
    unsigned long foreground; /* foreground pixel */
    unsigned long background; /* background pixel */
    int line_width;            /* line width (in pixels) */
```

```
        int line_style;              /* LineSolid, LineOnOffDash,
                                         LineDoubleDash */
        int cap_style;               /* CapNotLast, CapButt, CapRound,
                                         CapProjecting */
        int join_style;              /* JoinMiter, JoinRound, JoinBevel */
        int fill_style;              /* FillSolid, FillTiled,
                                         FillStippled FillOpaqueStippled*/
        int fill_rule;               /* EvenOddRule, WindingRule */
        int arc_mode;                /* ArcChord, ArcPieSlice */
        Pixmap tile;                 /* tile pixmap for tiling operations */
        Pixmap stipple;              /* stipple 1 plane pixmap for stippling */
        int ts_x_origin;             /* offset for tile or stipple operations */
        int ts_y_origin;
        Font font;                   /* default text font for text operations */
        int subwindow_mode;          /* ClipByChildren, IncludeInferiors */
        Bool graphics_exposures;     /* boolean, should exposures be generated */
        int clip_x_origin;           /* origin for clipping */
        int clip_y_origin;
        Pixmap clip_mask;            /* bitmap clipping; other calls for rects */
        int dash_offset;             /* patterned/dashed line information */
        char dashes;
} XGCValues;
```

The default GC values are:

Component	Default
function	GXcopy
plane_mask	All ones
foreground	0
background	1
line_width	0
line_style	LineSolid
cap_style	CapButt
join_style	JoinMiter
fill_style	FillSolid
fill_rule	EvenOddRule
arc_mode	ArcPieSlice
tile	Pixmap of unspecified size filled with foreground pixel (that is, client specified pixel if any, else 0) (subsequent changes to foreground do not affect this pixmap) stipple Pixmap of unspecified size filled with ones
ts_x_origin	0
ts_y_origin	0
font	<implementation dependent>
subwindow_mode	ClipByChildren
graphics_exposures	True
clip_x_origin	0
clip_y_origin	0
clip_mask	None
dash_offset	0
dashes	4 (that is, the list [4, 4])

Note that foreground and background are not set to any values likely to be useful in a window.

The function attributes of a GC are used when you update a section of a drawable (the destination) with bits from somewhere else (the source). The function in a GC defines how the new destination bits are to be computed from the source bits and the old destination bits. *GXcopy* is typically the most useful because it will work on a color display, but special applications may use other functions, particularly in concert with particular planes of a color display. The 16 GC functions, defined in <**X11/X.h**>, are:

Function Name	Value	Operation
GXclear	0x0	0
GXand	0x1	src AND dst
GXandReverse	0x2	src AND NOT dst
GXcopy	0x3	src
GXandInverted	0x4	(NOT src) AND dst
GXnoop	0x5	dst
GXxor	0x6	src XOR dst
GXor	0x7	src OR dst
GXnor	0x8	(NOT src) AND (NOT dst)
GXequiv	0x9	(NOT src) XOR dst
GXinvert	0xa	NOT dst
GXorReverse	0xb	src OR (NOT dst)
GXcopyInverted	0xc	NOT src
GXorInverted	0xd	(NOT src) OR dst
GXnand	0xe	(NOT src) OR (NOT dst)
GXset	0xf	1

Many graphics operations depend on either pixel values or planes in a GC. The planes attribute is of type long, and it specifies which planes of the destination are to be modified, one bit per plane. A monochrome display has only one plane and will be the least-significant bit of the word. As planes are added to the display hardware, they will occupy more significant bits in the plane mask.

In graphics operations, given a source and destination pixel, the result is computed bitwise on corresponding bits of the pixels. That is, a Boolean operation is performed in each bit plane. The plane_mask restricts the operation to a subset of planes. A macro constant *AllPlanes* can be used to refer to all planes of the screen simultaneously. The result is computed by the following:

```
((src FUNC dst) AND plane-mask) OR (dst AND (NOT plane-mask))
```

Range checking is not performed on the values for foreground, background or plane_mask. They are simply truncated to the appropriate number of bits. The line-width is measured in pixels and either can be greater than or equal to one (wide line) or can be the special value zero (thin line).

Wide lines are drawn centered on the path described by the graphics request. Unless otherwise specified by the join-style or cap-style, the bounding box of a wide line with endpoints [x1, y1], [x2, y2] and width w is a rectangle with vertices at the following real coordinates:

```
[x1-(w*sn/2), y1+(w*cs/2)], [x1+(w*sn/2), y1-(w*cs/2)],
[x2-(w*sn/2), y2+(w*cs/2)], [x2+(w*sn/2), y2-(w*cs/2)]
```

Here sn is the sine of the angle of the line, and cs is the cosine of the angle of the line. A pixel is part of the line and so is drawn if the center of the pixel is fully inside the bounding box (which is viewed as having infinitely thin edges). If the center of the pixel is exactly on the bounding box, it is part of the line if and only if the interior is immediately to its right (x increasing

direction). Pixels with centers on a horizontal edge are a special case and are part of the line if and only if the interior or the boundary is immediately below (y increasing direction) and the interior or the boundary is immediately to the right (x increasing direction).

Thin lines (zero line-width) are one-pixel-wide lines drawn using an unspecified, device-dependent algorithm. There are only two constraints on this algorithm.

1. If a line is drawn unclipped from [x1,y1] to [x2,y2] and if another line is drawn unclipped from [x1+dx,y1+dy] to [x2+dx,y2+dy], a point [x,y] is touched by drawing the first line if and only if the point [x+dx,y+dy] is touched by drawing the second line.

2. The effective set of points comprising a line cannot be affected by clipping. That is, a point is touched in a clipped line if and only if the point lies inside the clipping region and the point would be touched by the line when drawn unclipped.

A wide line drawn from [x1,y1] to [x2,y2] always draws the same pixels as a wide line drawn from [x2,y2] to [x1,y1], not counting cap-style and join-style. It is recommended that this property be true for thin lines, but this is not required. A line-width of zero may differ from a line-width of one in which pixels are drawn. This permits the use of many manufacturers' line drawing hardware, which may run many times faster than the more precisely specified wide lines.

In general, drawing a thin line will be faster than drawing a wide line of width one. However, because of their different drawing algorithms, thin lines may not mix well aesthetically with wide lines. If it is desirable to obtain precise and uniform results across all displays, a client should always use a line-width of one rather than a line-width of zero.

The line-style defines which sections of a line are drawn:

LineSolid	The full path of the line is drawn.
LineDoubleDash	The full path of the line is drawn, but the even dashes are filled differently than the odd dashes (see fill-style) with *CapButt* style used where even and odd dashes meet.
LineOnOffDash	Only the even dashes are drawn, and cap-style applies to all internal ends of the individual dashes, except *CapNotLast* is treated as *CapButt*.

The cap-style defines how the endpoints of a path are drawn:

CapNotLast	This is equivalent to *CapButt* except that for a line-width of zero the final endpoint is not drawn.
CapButt	The line is square at the endpoint (perpendicular to the slope of the line) with no projection beyond.
CapRound	The line has a circular arc with the diameter equal to the line-width, centered on the endpoint. (This is equivalent to *CapButt* for line-width of zero).
CapProjecting	The line is square at the end, but the path continues beyond the endpoint for a distance equal to half the line-width. (This is equivalent to *CapButt* for line-width of zero).

The join-style defines how corners are drawn for wide lines:

JoinMiter	The outer edges of two lines extend to meet at an angle. However, if the angle is less than 11 degrees, then a *JoinBevel* join-style is used instead.
JoinRound	The corner is a circular arc with the diameter equal to the line-width, centered on the joinpoint.
JoinBevel	The corner has *CapButt* endpoint styles with the triangular notch filled.

For a line with coincident endpoints (x1=x2, y1=y2), when the cap-style is applied to both endpoints, the semantics depends on the line-width and the cap-style:

CapNotLast	thin	The results are device-dependent, but the desired effect is that nothing is drawn.
CapButt	thin	The results are device-dependent, but the desired effect is that a single pixel is drawn.
CapRound	thin	The results are the same as for *CapButt*/thin.
CapProjecting	thin	The results are the same as for *CapButt*/thin.
CapButt	wide	Nothing is drawn.
CapRound	wide	The closed path is a circle, centered at the endpoint, and with the diameter equal to the line-width.
CapProjecting	wide	The closed path is a square, aligned with the coordinate axes, centered at the endpoint, and with the sides equal to the line-width.

For a line with coincident endpoints (x1=x2, y1=y2), when the join-style is applied at one or both endpoints, the effect is as if the line was removed from the overall path. However, if the total path consists of or is reduced to a single point joined with itself, the effect is the same as when the cap-style is applied at both endpoints.

The tile/stipple represents an infinite 2D plane, with the tile/stipple replicated in all dimensions. When that plane is superimposed on the drawable for use in a graphics operation, the upper left corner of some instance of the tile/stipple is at the coordinates within the drawable specified by the tile/stipple origin. The tile/stipple and clip origins are interpreted relative to the origin of whatever destination drawable is specified in a graphics request. The tile pixmap must have the same root and depth as the GC, or a *BadMatch* error results. The stipple pixmap must have depth one and must have the same root as the GC, or a *BadMatch* error results. For stipple operations where the fill-style is *FillStippled* but not *FillOpaqueStippled*, the stipple pattern is tiled in a single plane and acts as an additional clip mask to be ANDed with the clip-mask. Although some sizes may be faster to use than others, any size pixmap can be used for tiling or stippling.

The fill-style defines the contents of the source for line, text and fill requests. For all text and fill requests (for example, *XDrawText*, *XDrawText16*, *XFillRectangle*, *XFillPolygon* and *XFillArc*); for line requests with line-style *LineSolid* (for example, *XDrawLine*, *XDrawSegments*, *XDrawRectangle*, *XDrawArc*); and for the even dashes for line requests with line-style *LineOnOffDash* or *LineDoubleDash*, the following apply:

FillSolid	Foreground
FillTiled	Tile
FillOpaqueStippled	A tile with the same width and height as stipple, but with background everywhere stipple has a zero and with foreground everywhere stipple has a one
FillStippled	Foreground masked by stipple

When drawing lines with line-style *LineDoubleDash*, the odd dashes are controlled by the fill-style in the following manner:

FillSolid	Background
FillTiled	Same as for even dashes
FillOpaqueStippled	Same as for even dashes
FillStippled	Background masked by stipple

Storing a pixmap in a GC might or might not result in a copy being made. If the pixmap is later used as the destination for a graphics request, the change might or might not be reflected in the GC. If the pixmap is used simultaneously in a graphics request both as a destination and as a tile or stipple, the results are undefined.

For optimum performance, you should draw as much as possible with the same GC (without changing its components). The costs of changing GC components relative to using different GCs depend upon the display hardware and the server implementation. It is quite likely that some amount of GC information will be cached in display hardware and that such hardware can only cache a small number of GCs.

The dashes value is actually a simplified form of the more general patterns that can be set with *XSetDashes*. Specifying a value of N is equivalent to specifying the two-element list [N, N] in *XSetDashes*. The value must be non-zero, or a *BadValue* error results.

The clip-mask restricts writes to the destination drawable. If the clip-mask is set to a pixmap, it must have depth one and have the same root as the GC, or a *BadMatch* error results. If clip-mask is set to *None*, the pixels are always drawn regardless of the clip origin. The clip-mask also can be set by calling the *XSetClipRectangles* or *XSetRegion* functions. Only pixels where the clip-mask has a bit set to 1 are drawn. Pixels are not drawn outside the area covered by the clip-mask or where the clip-mask has a bit set to 0. The clip-mask affects all graphics requests. The clip-mask does not clip sources. The clip-mask origin is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

You can set the subwindow-mode to *ClipByChildren* or *IncludeInferiors*. For *ClipByChildren*, both source and destination windows are additionally clipped by all viewable *InputOutput* children. For *IncludeInferiors*, neither source nor destination window is clipped by inferiors. This will result in including subwindow contents in the source and drawing through subwindow boundaries of the destination. The use of *IncludeInferiors* on a window of one depth with mapped inferiors of differing depth is not illegal, but the semantics are undefined by the core protocol.

The fill-rule defines what pixels are inside (drawn) for paths given in *XFillPolygon* requests and can be set to *EvenOddRule* or *WindingRule*. For *EvenOddRule*, a point is inside if an infinite ray with the point as origin crosses the path an odd number of times. For *WindingRule*, a point is inside if an infinite ray with the point as origin crosses an unequal number of clockwise and counterclockwise directed path segments. A clockwise directed path segment is one that crosses the ray from left to right as observed from the point. A counterclockwise segment is one that crosses the ray from right to left as observed from the point. The case where a directed line segment is coincident with the ray is uninteresting because you can simply choose a different ray that is not coincident with a segment.

For both *EvenOddRule* and *WindingRule*, a point is infinitely small, and the path is an infinitely thin line. A pixel is inside if the center point of the pixel is inside and the center point is not on the boundary. If the center point is on the boundary, the pixel is inside if and only if the polygon interior is immediately to its right (x increasing direction). Pixels with centers on a horizontal edge are a special case and are inside if and only if the polygon interior is immediately below (y increasing direction).

The arc-mode controls filling in the *XFillArcs* function and can be set to *ArcPieSlice* or *ArcChord*. For *ArcPieSlice*, the arcs are pie-slice filled. For *ArcChord*, the arcs are chord filled.

The graphics-exposure flag controls *GraphicsExpose* event generation for *XCopyArea* and *XCopyPlane* requests (and any similar requests defined by extensions).

To create a new GC that is usable on a given screen with a depth of drawable, use *XCreateGC*.

```
GC XCreateGC(display, d, valuemask, values)
      Display *display;
      Drawable d;
      unsigned long valuemask;
      XGCValues *values;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

valuemask   Specifies which components in the GC are to be set using the information in the specified values structure. This argument is the bitwise inclusive OR of zero or more of the valid GC component mask bits.

values   Specifies any values as specified by the valuemask.

The *XCreateGC* function creates a graphics context and returns a GC. The GC can be used with any destination drawable having the same root and depth as the specified drawable. Use with other drawables results in a *BadMatch* error.

XCreateGC can generate *BadAlloc*, *BadDrawable*, *BadFont*, *BadMatch*, *BadPixmap* and *BadValue* errors.

To copy components from a source GC to a destination GC, use *XCopyGC*.

```
XCopyGC(display, src, valuemask, dest)
      Display *display;
      GC src, dest;
      unsigned long valuemask;
```

display   Specifies the connection to the X server.

src   Specifies the components of the source GC.

valuemask   Specifies which components in the GC are to be copied to the destination GC. This argument is the bitwise inclusive OR of zero or more of the valid GC component mask bits.

dest   Specifies the destination GC.

The *XCopyGC* function copies the specified components from the source GC to the destination GC. The source and destination GCs must have the same root and depth, or a *BadMatch* error results. The valuemask specifies which component to copy, as for *XCreateGC*.

XCopyGC can generate *BadAlloc*, *BadGC* and *BadMatch* errors.

To change the components in a given GC, use *XChangeGC*.

```
XChangeGC(display, gc, valuemask, values)
      Display *display;
      GC gc;
      unsigned long valuemask;
      XGCValues *values;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

valuemask   Specifies which components in the GC are to be changed using information in
                     the specified values structure. This argument is the bitwise inclusive OR of
                     zero or more of the valid GC component mask bits.

values   Specifies any values as specified by the valuemask.

The *XChangeGC* function changes the components specified by valuemask for the specified GC.
The values argument contains the values to be set. The values and restrictions are the same as
for *XCreateGC*. Changing the clip-mask overrides any previous *XSetClipRectangles* request on
the context. Changing the dash-offset or dash-list overrides any previous *XSetDashes* request on
the context. The order in which components are verified and altered is server-dependent. If an
error is generated, a subset of the components may have been altered.

XChangeGC can generate *BadAlloc*, *BadFont*, *BadGC*, *BadMatch*, *BadPixmap* and *BadValue* errors.

To obtain components of a given GC, use *XGetGCValues*.

```
Status XGetGCValues(display, gc, valuemask, values_return)
      Display *display;
      GC gc;
      unsigned long valuemask;
      XGCValues *values_return;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

valuemask   Specifies which components in the GC are to be returned in the values_return
                     argument. This argument is the bitwise inclusive OR of zero or more of the
                     valid GC component mask bits.

values_return   Returns the GC values in the specified *XGCValues* structure.

The *XGetGCValues* function returns the components specified by valuemask for the specified
GC. If the valuemask contains a valid set of GC mask bits (*GCFunction*, *GCPlaneMask*,
GCForeground, *GCBackground*, *GCLineWidth*, *GCLineStyle*, *GCCapStyle*, *GCJoinStyle*, *GCFillStyle*,
GCFillRule, *GCTile*, *GCStipple*, *GCTileStipXOrigin*, *GCTileStipYOrigin*, *GCFont*,
GCSubwindowMode, *GCGraphicsExposures*, *GCClipXOrigin*, *GCLipYOrigin*, *GCDashOffset* or
GCArcMode) and no error occur, *XGetGCValues* sets the requested components in values_return
and returns a non-zero status. Otherwise, it returns a zero status. Note that the clip-mask and
dash-list (represented by the *GCClipMask* and *GCDashList* bits, respectively, in the valuemask)
cannot be requested. Also note that an invalid resource ID (with one or more of the three most-
significant bits set to one) will be returned for *GCFont*, *GCTile* and *GCStipple* if the component
has never been explicitly set by the client.

To free a given GC, use *XFreeGC*.

```
XFreeGC(display, gc)
      Display *display;
      GC gc;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

The *XFreeGC* function destroys the specified GC as well as all the associated storage.

XFreeGC can generate a *BadGC* error.

To obtain the *GContext* resource ID for a given GC, use *XGContextFromGC*.

```
GContext XGContextFromGC(gc)
      GC gc;
```

gc   Specifies the GC for which you want the resource ID.

Xlib normally defers sending changes to the components of a GC to the server until a graphics function is actually called with that GC. This permits batching of component changes into a single server request. In some circumstances, however, it may be necessary for the client to explicitly force sending of the changes to the server. An example might be when a protocol extension uses the GC indirectly, in such a way that the extension interface cannot know what GC will be used. To force sending of GC component changes, use *XFlushGC*.

```
void XFlushGC(display, gc)
      Display *display;
      GC gc;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

8.2   Using GC Convenience Routines

This section discusses how to set the:

- foreground, background, plane mask or function components

- line attributes and dashes components

- fill style and fill rule components

- fill tile and stipple components

- font component

- clip region component

- arc mode, subwindow mode and graphics exposure components.

8.2.1   Setting the Foreground, Background, Function, or Plane Mask

To set the foreground, background, plane mask and function components for a given GC, use *XSetState*.

```
XSetState(display, gc, foreground, background,
        function, plane_mask)
    Display *display;
    GC gc;
    unsigned long foreground, background;
    int function;
    unsigned long plane_mask;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

foreground   Specifies the foreground you want to set for the specified GC.

background   Specifies the background you want to set for the specified GC.

function   Specifies the function you want to set for the specified GC.

plane_mask   Specifies the plane mask.

XSetState can generate *BadAlloc*, *BadGC* and *BadValue* errors.

To set the foreground of a given GC, use *XSetForeground*.

```
XSetForeground(display, gc, foreground)
    Display *display;
    GC gc;
    unsigned long foreground;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

foreground   Specifies the foreground you want to set for the specified GC.

XSetForeground can generate *BadAlloc* and *BadGC* errors.

To set the background of a given GC, use *XSetBackground*.

```
XSetBackground(display, gc, background)
      Display *display;
      GC gc;
      unsigned long background;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

background   Specifies the background you want to set for the specified GC.

XSetBackground can generate *BadAlloc* and *BadGC* errors.

To set the display function in a given GC, use *XSetFunction*.

```
XSetFunction(display, gc, function)
      Display *display;
      GC gc;
      int function;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

function   Specifies the function you want to set for the specified GC.

XSetFunction can generate *BadAlloc*, *BadGC* and *BadValue* errors.

To set the plane mask of a given GC, use *XSetPlaneMask*.

```
XSetPlaneMask(display, gc, plane_mask)
      Display *display;
      GC gc;
      unsigned long plane_mask;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

plane_mask   Specifies the plane mask.

XSetPlaneMask can generate *BadAlloc* and *BadGC* errors.

8.2.2   Setting the Line Attributes and Dashes

To set the line drawing components of a given GC, use *XSetLineAttributes*.

```
XSetLineAttributes(display, gc, line_width,
         line_style, cap_style, join_style)
      Display *display;
      GC gc;
      unsigned int line_width;
      int line_style;
      int cap_style;
      int join_style;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

line_width   Specifies the line-width you want to set for the specified GC.

line_style   Specifies the line-style you want to set for the specified GC. You can pass
                    LineSolid, *LineOnOffDash* or *LineDoubleDash*.

cap_style   Specifies the line-style and cap-style you want to set for the specified GC. You
                    can pass *CapNotLast*, *CapButt*, *CapRound* or *CapProjecting*.

join_style   Specifies the line join-style you want to set for the specified GC. You can pass
                    JoinMiter, *JoinRound* or *JoinBevel*.

XSetLineAttributes can generate *BadAlloc*, *BadGC* and *BadValue* errors.

To set the dash-offset and dash-list for dashed line styles of a given GC, use *XSetDashes*.

```
XSetDashes(display, gc, dash_offset, dash_list, n)
        Display *display;
        GC gc;
        int dash_offset;
        char dash_list[];
        int n;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

dash_offset   Specifies the phase of the pattern for the dashed line-style you want to set for
                    the specified GC.

dash_list   Specifies the dash-list for the dashed line-style you want to set for the
                    specified GC.

n   Specifies the number of elements in dash_list.

The *XSetDashes* function sets the dash-offset and dash-list attributes for dashed line styles in the
specified GC. There must be at least one element in the specified dash_list, or a *BadValue* error
results. The initial and alternating elements (second, fourth, and so on) of the dash_list are the
even dashes, and the others are the odd dashes. Each element specifies a dash length in pixels.
All of the elements must be non-zero, or a *BadValue* error results. Specifying an odd-length list is
equivalent to specifying the same list concatenated with itself to produce an even-length list.

The dash-offset defines the phase of the pattern, specifying how many pixels into the dash-list
the pattern should actually begin in any single graphics request. Dashing is continuous through
path elements combined with a join-style but is reset to the dash-offset between each sequence
of joined lines.

The unit of measure for dashes is the same for the ordinary coordinate system. Ideally, a dash
length is measured along the slope of the line, but implementations are only required to match
this ideal for horizontal and vertical lines. Failing the ideal semantics, it is suggested that the
length be measured along the major axis of the line. The major axis is defined as the x axis for
lines drawn at an angle of between −45 and +45 degrees or between 135 and 225 degrees from
the x axis. For all other lines, the major axis is the y axis.

XSetDashes can generate *BadAlloc*, *BadGC* and *BadValue* errors.

8.2.3   Setting the Fill Style and Fill Rule

To set the fill-style of a given GC, use *XSetFillStyle*.

```
XSetFillStyle(display, gc, fill_style)
      Display *display;
      GC gc;
      int fill_style;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

fill_style   Specifies the fill-style you want to set for the specified GC. You can pass
                        FillSolid, *FillTiled*, *FillStippled* or *FillOpaqueStippled*.

XSetFillStyle can generate *BadAlloc*, *BadGC* and *BadValue* errors.

To set the fill-rule of a given GC, use *XSetFillRule*.

```
XSetFillRule(display, gc, fill_rule)
      Display *display;
      GC gc;
      int fill_rule;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

fill_rule   Specifies the fill-rule you want to set for the specified GC. You can pass
                        EvenOddRule or *WindingRule*.

                        XSetFillRule can generate *BadAlloc*, *BadGC* and *BadValue* errors.

8.2.4   Setting the Fill Tile and Stipple

Some displays have hardware support for tiling or stippling with patterns of specific sizes.
Tiling and stippling operations that restrict themselves to those specific sizes run much faster
than such operations with arbitrary size patterns. Xlib provides functions that you can use to
determine the best size, tile or stipple for the display as well as to set the tile or stipple shape and
the tile or stipple origin.

To obtain the best size of a tile, stipple, or cursor, use *XQueryBestSize*.

```
Status XQueryBestSize(display, class, which_screen,
        width, height, width_return, height_return)
      Display *display;
      int class;
      Drawable which_screen;
      unsigned int width, height;
      unsigned int *width_return, *height_return;
```

display   Specifies the connection to the X server.

class   Specifies the class that you are interested in. You can pass *TileShape*,
                        CursorShape or *StippleShape*.

which_screen   Specifies any drawable on the screen.

width
height   Specify the width and height.

width_return
height_return   Return the width and height of the object best supported by the display hardware.

The *XQueryBestSize* function returns the best or closest size to the specified size. For *CursorShape*, this is the largest size that can be fully displayed on the screen specified by which_screen. For *TileShape*, this is the size that can be tiled fastest. For *StippleShape*, this is the size that can be stippled fastest. For *CursorShape*, the drawable indicates the desired screen. For *TileShape* and *StippleShape*, the drawable indicates the screen and possibly the window class and depth. An *InputOnly* window cannot be used as the drawable for *TileShape* or *StippleShape*, or a *BadMatch* error results.

XQueryBestSize can generate *BadDrawable*, *BadMatch* and *BadValue* errors.

To obtain the best fill tile shape, use *XQueryBestTile*.

```
Status XQueryBestTile(display, which_screen, width,
        height, width_return, height_return)
     Display *display;
     Drawable which_screen;
     unsigned int width, height;
     unsigned int *width_return, *height_return;
```

display   Specifies the connection to the X server.

which_screen   Specifies any drawable on the screen.

width
height   Specify the width and height.

width_return
height_return   Return the width and height of the object best supported by the display hardware.

The *XQueryBestTile* function returns the best or closest size, that is, the size that can be tiled fastest on the screen specified by which_screen. The drawable indicates the screen and possibly the window class and depth. If an *InputOnly* window is used as the drawable, a *BadMatch* error results.

XQueryBestTile can generate *BadDrawable* and *BadMatch* errors.

To obtain the best stipple shape, use *XQueryBestStipple*.

```
Status XQueryBestStipple(display, which_screen, width,
        height, width_return, height_return)
     Display *display;
     Drawable which_screen;
     unsigned int width, height;
     unsigned int *width_return, *height_return;
```

display   Specifies the connection to the X server.

which_screen   Specifies any drawable on the screen.

width
height   Specify the width and height.

width_return
height_return   Return the width and height of the object best supported by the display hardware.

The *XQueryBestStipple* function returns the best or closest size, that is, the size that can be stippled fastest on the screen specified by which_screen. The drawable indicates the screen and possibly the window class and depth. If an *InputOnly* window is used as the drawable, a *BadMatch* error results.

XQueryBestStipple can generate *BadDrawable* and *BadMatch* errors.

To set the fill tile of a given GC, use *XSetTile*.

```
XSetTile(display, gc, tile)
      Display *display;
      GC gc;
      Pixmap tile;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

tile   Specifies the fill tile you want to set for the specified GC.

The tile and GC must have the same depth, or a *BadMatch* error results.

XSetTile can generate *BadAlloc*, *BadGC*, *BadMatch* and *BadPixmap* errors.

To set the stipple of a given GC, use *XSetStipple*.

```
XSetStipple(display, gc, stipple)
      Display *display;
      GC gc;
      Pixmap stipple;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

stipple   Specifies the stipple you want to set for the specified GC.

The stipple must have a depth of one, or a *BadMatch* error results.

XSetStipple can generate *BadAlloc*, *BadGC*, *BadMatch* and *BadPixmap* errors.

To set the tile or stipple origin of a given GC, use *XSetTSOrigin*.

```
XSetTSOrigin(display, gc, ts_x_origin, ts_y_origin)
      Display *display;
      GC gc;
      int ts_x_origin, ts_y_origin;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

ts_x_origin
ts_y_origin   Specify the x and y coordinates of the tile and stipple origin.

When graphics requests call for tiling or stippling, the parent's origin will be interpreted relative to whatever destination drawable is specified in the graphics request.

XSetTSOrigin can generate *BadAlloc* and *BadGC* error.

8.2.5   **Setting the Current Font**

To set the current font of a given GC, use *XSetFont*.

```
XSetFont(display, gc, font)
      Display *display;
      GC gc;
      Font font;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

font   Specifies the font.

XSetFont can generate *BadAlloc*, *BadFont* and *BadGC* errors.

8.2.6   **Setting the Clip Region**

Xlib provides functions that you can use to set the clip-origin and the clip-mask or set the clip-mask to a list of rectangles.

To set the clip-origin of a given GC, use *XSetClipOrigin*.

```
XSetClipOrigin(display, gc, clip_x_origin, clip_y_origin)
      Display *display;
      GC gc;
      int clip_x_origin, clip_y_origin;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

clip_x_origin
clip_y_origin   Specify the x and y coordinates of the clip-mask origin.

The clip-mask origin is interpreted relative to the origin of whatever destination drawable is specified in the graphics request.

XSetClipOrigin can generate *BadAlloc* and *BadGC* errors.

To set the clip-mask of a given GC to the specified pixmap, use *XSetClipMask*.

```
XSetClipMask(display, gc, pixmap)
      Display *display;
      GC gc;
      Pixmap pixmap;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

pixmap   Specifies the pixmap or *None*.

If the clip-mask is set to *None*, the pixels are are always drawn (regardless of the clip-origin).

XSetClipMask can generate *BadAlloc*, *BadGC*, *BadMatch* and *BadPixmap* errors.

To set the clip-mask of a given GC to the specified list of rectangles, use *XSetClipRectangles*.

```
XSetClipRectangles(display, gc, clip_x_origin,
        clip_y_origin, rectangles, n, ordering)
    Display *display;
    GC gc;
    int clip_x_origin, clip_y_origin;
    XRectangle rectangles[];
    int n;
    int ordering;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

clip_x_origin
clip_y_origin   Specify the x and y coordinates of the clip-mask origin.

rectangles   Specifies an array of rectangles that define the clip-mask.

n   Specifies the number of rectangles.

ordering   Specifies the ordering relations on the rectangles. You can pass *Unsorted*,
                   YSorted, *YXSorted* or *YXBanded*.

The *XSetClipRectangles* function changes the clip-mask in the specified GC to the specified list of
rectangles and sets the clip origin. The output is clipped to remain contained within the
rectangles. The clip-origin is interpreted relative to the origin of whatever destination drawable
is specified in a graphics request. The rectangle coordinates are interpreted relative to the clip-
origin. The rectangles should be nonintersecting, or the graphics results will be undefined. Note
that the list of rectangles can be empty, which effectively disables output. This is the opposite of
passing *None* as the clip-mask in *XCreateGC*, *XChangeGC* and *XSetClipMask*.

If known by the client, ordering relations on the rectangles can be specified with the ordering
argument. This may provide faster operation by the server. If an incorrect ordering is specified,
the X server may generate a *BadMatch* error, but it is not required to do so. If no error is
generated, the graphics results are undefined. *Unsorted* means the rectangles are in arbitrary
order. *YSorted* means that the rectangles are non-decreasing in their Y origin. *YXSorted*
additionally constrains *YSorted* order in that all rectangles with an equal Y origin are non-
decreasing in their X origin. *YXBanded* additionally constrains *YXSorted* by requiring that, for
every possible Y scanline, all rectangles that include that scanline have an identical Y origins and
Y extents.

XSetClipRectangles can generate *BadAlloc*, *BadGC*, *BadMatch* and *BadValue* errors.

Xlib provides a set of basic functions for performing region arithmetic. For information about
these functions, see Section 17.5.

8.2.7   Setting the Arc Mode, Subwindow Mode and Graphics Exposure

To set the arc mode of a given GC, use *XSetArcMode*.

```
XSetArcMode(display, gc, arc_mode)
    Display *display;
    GC gc;
    int arc_mode;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

arc_mode   Specifies the arc mode. You can pass *ArcChord* or *ArcPieSlice*.

XSetArcMode can generate *BadAlloc*, *BadGC* and *BadValue* errors.

To set the subwindow mode of a given GC, use *XSetSubwindowMode*.

```
XSetSubwindowMode(display, gc, subwindow_mode)
      Display *display;
      GC gc;
      int subwindow_mode;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

subwindow_mode Specifies the subwindow mode. You can pass *ClipByChildren* or *IncludeInferiors*.

XSetSubwindowMode can generate *BadAlloc*, *BadGC* and *BadValue* errors.

To set the graphics-exposures flag of a given GC, use *XSetGraphicsExposures*.

```
XSetGraphicsExposures(display, gc, graphics_exposures)
      Display *display;
      GC gc;
      Bool graphics_exposures;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

graphics_exposures Specifies a Boolean value that indicates whether you want *GraphicsExpose* and *NoExpose* events to be reported when calling *XCopyArea* and *XCopyPlane* with this GC.

XSetGraphicsExposures can generate *BadAlloc*, *BadGC* and *BadValue* errors.

Graphics Functions

Once you have established a connection to a display, you can use the Xlib graphics functions to:

- clear and copy areas

- draw points, lines, rectangles and arcs

- fill areas

- manipulate fonts

- draw text

- transfer images between clients and the server.

If the same drawable and GC is used for each call, Xlib batches back-to-back calls to *XDrawPoint*, *XDrawLine*, *XDrawRectangle*, *XFillArc* and *XFillRectangle*. Note that this reduces the total number of requests sent to the server.

9.1 Clearing Areas

Xlib provides functions that you can use to clear an area or the entire window. Because pixmaps do not have defined backgrounds, they cannot be filled by using the functions described in this section. Instead, to accomplish an analogous operation on a pixmap, you should use *XFillRectangle*, which sets the pixmap to a known value.

To clear a rectangular area of a given window, use *XClearArea*.

```
XClearArea(display, w, x, y, width,
        height, exposures)
    Display *display;
    Window w;
    int x, y;
    unsigned int width, height;
    Bool exposures;
```

display   Specifies the connection to the X server.

w   Specifies the window.

x
y   Specify the x and y coordinates, which are relative to the origin of the window and specify the upper-left corner of the rectangle.

width
height   Specify the width and height, which are the dimensions of the rectangle.

exposures   Specifies a Boolean value that indicates if *Expose* events are to be generated.

The *XClearArea* function paints a rectangular area in the specified window according to the specified dimensions with the window's background pixel or pixmap. The subwindow-mode effectively is *ClipByChildren*. If width is zero, it is replaced with the current width of the window minus x. If height is zero, it is replaced with the current height of the window minus y. If the window has a defined background tile, the rectangle clipped by any children is filled with this tile. If the window has background *None*, the contents of the window are not changed. In either case, if exposures is *True*, one or more *Expose* events are generated for regions of the rectangle that are either visible or are being retained in a backing store. If you specify a window whose class is *InputOnly*, a *BadMatch* error results.

XClearArea can generate *BadMatch*, *BadValue* and *BadWindow* errors.

To clear the entire area in a given window, use *XClearWindow*.

```
XClearWindow(display, w)
    Display *display;
    Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XClearWindow* function clears the entire area in the specified window and is equivalent to *XClearArea* (display, w, 0, 0, 0, 0, *False*). If the window has a defined background tile, the rectangle is tiled with a plane-mask of all ones and *GXcopy* function. If the window has background *None*, the contents of the window are not changed. If you specify a window whose class is *InputOnly*, a *BadMatch* error results.

XClearWindow can generate *BadMatch* and *BadWindow* errors.

9.2   Copying Areas

Xlib provides functions that you can use to copy an area or a bit plane.

To copy an area between drawables of the same root and depth, use *XCopyArea*.

```
XCopyArea(display, src, dest, gc, src_x,
        src_y, width, height,  dest_x, dest_y)
    Display *display;
    Drawable src, dest;
    GC gc;
    int src_x, src_y;
    unsigned int width, height;
    int dest_x, dest_y;
```

display   Specifies the connection to the X server.

src
dest   Specify the source and destination rectangles to be combined.

gc   Specifies the GC.

src_x
src_y   Specify the x and y coordinates, which are relative to the origin of the source
                   rectangle and specify its upper-left corner.

width
height   Specify the width and height, which are the dimensions of both the source and
                   destination rectangles.

dest_x
dest_y   Specify the x and y coordinates, which are relative to the origin of the
                   destination rectangle and specify its upper-left corner.

The *XCopyArea* function combines the specified rectangle of src with the specified rectangle of
dest. The drawables must have the same root and depth, or a *BadMatch* error results.

If regions of the source rectangle are obscured and have not been retained in backing store or if
regions outside the boundaries of the source drawable are specified, those regions are not
copied. Instead, the following occurs on all corresponding destination regions that are either
visible or are retained in backing store. If the destination is a window with a background other
than *None*, corresponding regions of the destination are tiled with that background (with plane-
mask of all ones and *GXcopy* function). Regardless of tiling or whether the destination is a
window or a pixmap, if graphics-exposures is *True*, then *GraphicsExpose* events for all
corresponding destination regions are generated. If graphics-exposures is *True* but no
GraphicsExpose events are generated, a *NoExpose* event is generated. Note that by default
graphics-exposures is *True* in new GCs.

This function uses these GC components: function, plane-mask, subwindow-mode, graphics-
exposures, clip-x-origin, clip-y-origin and clip-mask.

XCopyArea can generate *BadDrawable*, *BadGC* and *BadMatch* errors.

To copy a single bit plane of a given drawable, use *XCopyPlane*.

```
XCopyPlane(display, src, dest, gc, src_x,
        src_y, width, height, dest_x,
        dest_y, plane)
    Display *display;
    Drawable src, dest;
    GC gc;
    int src_x, src_y;
    unsigned int width, height;
    int dest_x, dest_y;
    unsigned long plane;
```

display   Specifies the connection to the X server.

src
dest   Specify the source and destination rectangles to be combined.

gc   Specifies the GC.

src_x
src_y   Specify the x and y coordinates, which are relative to the origin of the source
                   rectangle and specify its upper-left corner.

width
height   Specify the width and height, which are the dimensions of both the source and
                   destination rectangles.

dest_x
dest_y   Specify the x and y coordinates, which are relative to the origin of the
                   destination rectangle and specify its upper-left corner.

plane   Specifies the bit plane. You must set exactly one bit to 1.

The *XCopyPlane* function uses a single bit plane of the specified source rectangle combined with
the specified GC to modify the specified rectangle of dest. The drawables must have the same
root but need not have the same depth. If the drawables do not have the same root, a *BadMatch*
error results. If plane does not have exactly one bit set to 1 and the values of planes must be less
than 2^n, where *n* is the depth of src, a *BadValue* error results.

Effectively, *XCopyPlane* forms a pixmap of the same depth as the rectangle of dest and with a
size specified by the source region. It uses the foreground/background pixels in the GC
(foreground everywhere the bit plane in src contains a bit set to 1, background everywhere the
bit plane in src contains a bit set to 0) and the equivalent of a *CopyArea* protocol request is
performed with all the same exposure semantics. This can also be thought of as using the
specified region of the source bit plane as a stipple with a fill-style of *FillOpaqueStippled* for filling
a rectangular area of the destination.

This function uses these GC components: function, plane-mask, foreground, background,
subwindow-mode, graphics-exposures, clip-x-origin, clip-y-origin and clip-mask.

XCopyPlane can generate *BadDrawable*, *BadGC*, *BadMatch* and *BadValue* errors.

9.3   Drawing Points, Lines, Rectangles and Arcs

Xlib provides functions that you can use to draw:

- a single point or multiple points

- a single line or multiple lines

- a single rectangle or multiple rectangles

- a single arc or multiple arcs.

Some of the functions described in the following sections use these structures:

```
typedef struct {
      short x1, y1, x2, y2;
} XSegment;

typedef struct {
      short x, y;
} XPoint;

typedef struct {
      short x, y;
      unsigned short width, height;
} XRectangle;

typedef struct {
      short x, y;
      unsigned short width, height;
      short angle1, angle2;                /* Degrees * 64 */
} XArc;
```

All x and y members are signed integers. The width and height members are 16-bit unsigned integers. You should be careful not to generate coordinates and sizes out of the 16-bit ranges, because the protocol only has 16-bit fields for these values.

9.3.1   Drawing Single and Multiple Points

To draw a single point in a given drawable, use *XDrawPoint*.

```
XDrawPoint(display, d, gc, x, y)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x

y   Specify the x and y coordinates where you want the point drawn.

To draw multiple points in a given drawable, use *XDrawPoints*.

```
XDrawPoints(display, d, gc, points, npoints, mode)
      Display *display;
      Drawable d;
      GC gc;
      XPoint *points;
      int npoints;
      int mode;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

points   Specifies an array of points.

npoints   Specifies the number of points in the array.

mode   Specifies the coordinate mode. You can pass *CoordModeOrigin* or
                     CoordModePrevious.

The *XDrawPoint* function uses the foreground pixel and function components of the GC to draw
a single point into the specified drawable; *XDrawPoints* draws multiple points this way.
CoordModeOrigin treats all coordinates as relative to the origin, and *CoordModePrevious* treats all
coordinates after the first as relative to the previous point. *XDrawPoints* draws the points in the
order listed in the array.

Both functions use these GC components: function, plane-mask, foreground, subwindow-mode,
clip-x-origin, clip-y-origin and clip-mask.

XDrawPoint can generate *BadDrawable*, *BadGC* and *BadMatch* errors. *XDrawPoints* can generate
BadDrawable, *BadGC*, *BadMatch* and *BadValue* errors.

9.3.2   Drawing Single and Multiple Lines

To draw a single line between two points in a given drawable, use *XDrawLine*.

```
XDrawLine(display, d, gc, x1, y1, x2, y2)
      Display *display;
      Drawable d;
      GC gc;
      int x1, y1, x2, y2;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x1
y1
x2
y2   Specify the points (x1, y1) and (x2, y2) to be connected.

To draw multiple lines in a given drawable, use *XDrawLines*.

```
XDrawLines(display, d, gc, points, npoints, mode)
      Display *display;
      Drawable d;
      GC gc;
      XPoint *points;
      int npoints;
      int mode;
```

display	Specifies the connection to the X server.
d	Specifies the drawable.
gc	Specifies the GC.
points	Specifies an array of points.
npoints	Specifies the number of points in the array.
mode	Specifies the coordinate mode. You can pass *CoordModeOrigin* or *CoordModePrevious*.

To draw multiple, unconnected lines in a given drawable, use *XDrawSegments*.

```
XDrawSegments(display, d, gc, segments, nsegments)
      Display *display;
      Drawable d;
      GC gc;
      XSegment *segments;
      int nsegments;
```

display	Specifies the connection to the X server.
d	Specifies the drawable.
gc	Specifies the GC.
segments	Specifies an array of segments.
nsegments	Specifies the number of segments in the array.

The *XDrawLine* function uses the components of the specified GC to draw a line between the specified set of points (x1, y1) and (x2, y2). It does not perform joining at coincident endpoints. For any given line, *XDrawLine* does not draw a pixel more than once. If lines intersect, the intersecting pixels are drawn multiple times.

The *XDrawLines* function uses the components of the specified GC to draw npoints–1 lines between each pair of points (point[i], point[i+1]) in the array of *XPoint* structures. It draws the lines in the order listed in the array. The lines join correctly at all intermediate points, and if the first and last points coincide, the first and last lines also join correctly. For any given line, *XDrawLines* does not draw a pixel more than once. If thin (zero line-width) lines intersect, the intersecting pixels are drawn multiple times. If wide lines intersect, the intersecting pixels are drawn only once, as though the entire *PolyLine* protocol request were a single, filled shape. *CoordModeOrigin* treats all coordinates as relative to the origin, and *CoordModePrevious* treats all coordinates after the first as relative to the previous point.

The *XDrawSegments* function draws multiple, unconnected lines. For each segment, *XDrawSegments* draws a line between (x1, y1) and (x2, y2). It draws the lines in the order listed in the array of *XSegment* structures and does not perform joining at coincident endpoints. For any given line, *XDrawSegments* does not draw a pixel more than once. If lines intersect, the intersecting pixels are drawn multiple times.

All three functions use these GC components: function, plane-mask, line-width, line-style, cap-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. The *XDrawLines* function also uses the join-style GC component. All three functions also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, tile-stipple-y-origin, dash-offset and dash-list.

XDrawLine, *XDrawLines* and *XDrawSegments* can generate *BadDrawable*, *BadGC* and *BadMatch* errors. *XDrawLines* also can generate *BadValue* errors.

9.3.3   Drawing Single and Multiple Rectangles

To draw the outline of a single rectangle in a given drawable, use *XDrawRectangle*.

```
XDrawRectangle(display, d, gc, x, y,
        width, height)
    Display *display;
    Drawable d;
    GC gc;
    int x, y;
    unsigned int width, height;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x

y   Specify the x and y coordinates, which specify the upper-left corner of the rectangle.

width

height   Specify the width and height, which specify the dimensions of the rectangle.

To draw the outline of multiple rectangles in a given drawable, use *XDrawRectangles*.

```
XDrawRectangles(display, d, gc, rectangles,
        nrectangles)
    Display *display;
    Drawable d;
    GC gc;
    XRectangle rectangles[];
    int nrectangles;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

rectangles   Specifies an array of rectangles.

nrectangles   Specifies the number of rectangles in the array.

The *XDrawRectangle* and *XDrawRectangles* functions draw the outlines of the specified rectangle or rectangles as if a five-point *PolyLine* protocol request were specified for each rectangle:

```
[x,y] [x+width,y] [x+width,y+height] [x,y+height] [x,y]
```

For the specified rectangle or rectangles, these functions do not draw a pixel more than once. *XDrawRectangles* draws the rectangles in the order listed in the array. If rectangles intersect, the

intersecting pixels are drawn multiple times.

Both functions use these GC components: function, plane-mask, line-width, line-style, cap-style, join-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, tile-stipple-y-origin, dash-offset and dash-list.

XDrawRectangle and *XDrawRectangles* can generate *BadDrawable*, *BadGC* and *BadMatch* errors.

9.3.4   Drawing Single and Multiple Arcs

To draw a single arc in a given drawable, use *XDrawArc*.

```
XDrawArc(display, d, gc, x, y, width,
        height, angle1, angle2)
    Display *display;
    Drawable d;
    GC gc;
    int x, y;
    unsigned int width, height;
    int angle1, angle2;
```

display	Specifies the connection to the X server.
d	Specifies the drawable.
gc	Specifies the GC.
x	
y	Specify the x and y coordinates, which are relative to the origin of the drawable and specify the upper-left corner of the bounding rectangle.
width	
height	Specify the width and height, which are the major and minor axes of the arc.
angle1	Specifies the start of the arc relative to the three-o'clock position from the center, in units of degrees * 64.
angle2	Specifies the path and extent of the arc relative to the start of the arc, in units of degrees * 64.

To draw multiple arcs in a given drawable, use *XDrawArcs*.

```
XDrawArcs(display, d, gc, arcs, narcs)
    Display *display;
    Drawable d;
    GC gc;
    XArc *arcs;
    int narcs;
```

display	Specifies the connection to the X server.
d	Specifies the drawable.
gc	Specifies the GC.
arcs	Specifies an array of arcs.
narcs	Specifies the number of arcs in the array.

XDrawArc draws a single circular or elliptical arc, and *XDrawArcs* draws multiple circular or elliptical arcs. Each arc is specified by a rectangle and two angles. The center of the circle or

ellipse is the center of the rectangle, and the major and minor axes are specified by the width and height. Positive angles indicate counterclockwise motion, and negative angles indicate clockwise motion. If the magnitude of angle2 is greater than 360 degrees, *XDrawArc* or *XDrawArcs* truncates it to 360 degrees.

For an arc specified as [*x, y, width, height, angle*1, *angle*2], the origin of the major and minor axes is at [$x + \frac{width}{2}$, $y + \frac{height}{2}$], and the infinitely thin path describing the entire circle or ellipse intersects the horizontal axis at [$x, y + \frac{height}{2}$] and [$x + width, y + \frac{height}{2}$] and intersects the vertical axis at [$x + \frac{width}{2}, y$] and [$x + \frac{width}{2}, y + height$]. These coordinates can be fractional and so are not truncated to discrete coordinates. The path should be defined by the ideal mathematical path. For a wide line with line-width lw, the bounding outlines for filling are given by the two infinitely thin paths consisting of all points whose perpendicular distance from the path of the circle/ellipse is equal to lw/2 (which may be a fractional value). The cap-style and join-style are applied the same as for a line corresponding to the tangent of the circle/ellipse at the endpoint.

For an arc specified as [*x, y, width, height, angle*1, *angle*2], the angles must be specified in the effectively skewed coordinate system of the ellipse (for a circle, the angles and coordinate systems are identical). The relationship between these angles and angles expressed in the normal coordinate system of the screen (as measured with a protractor) is as follows:

$$\text{skewed-angle} = atan\left[\tan(\text{normal-angle}) * \frac{width}{height}\right] + adjust$$

The skewed-angle and normal-angle are expressed in radians (rather than in degrees scaled by 64) in the range [0, 2π] and where atan returns a value in the range [$-\frac{\pi}{2}$, $\frac{\pi}{2}$] and adjust is:

0   for normal-angle in the range [0, $\frac{\pi}{2}$]

π for normal-angle in the range [$\frac{\pi}{2}$, $\frac{3\pi}{2}$]

2π for normal-angle in the range [$\frac{3\pi}{2}$, 2π]

For any given arc, *XDrawArc* and *XDrawArcs* do not draw a pixel more than once. If two arcs join correctly and if the line-width is greater than zero and the arcs intersect, *XDrawArc* and *XDrawArcs* do not draw a pixel more than once. Otherwise, the intersecting pixels of intersecting arcs are drawn multiple times. Specifying an arc with one endpoint and a clockwise extent draws the same pixels as specifying the other endpoint and an equivalent counterclockwise extent, except as it affects joins.

If the last point in one arc coincides with the first point in the following arc, the two arcs will join correctly. If the first point in the first arc coincides with the last point in the last arc, the two arcs will join correctly. By specifying one axis to be zero, a horizontal or vertical line can be drawn. Angles are computed based solely on the coordinate system and ignore the aspect ratio.

Both functions use these GC components: function, plane-mask, line-width, line-style, cap-style, join-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, tile-stipple-y-origin, dash-offset and dash-list.

XDrawArc and *XDrawArcs* can generate *BadDrawable*, *BadGC* and *BadMatch* errors.

9.4 Filling Areas

Xlib provides functions that you can use to fill:

- a single rectangle or multiple rectangles
- a single polygon
- a single arc or multiple arcs.

9.4.1 Filling Single and Multiple Rectangles

To fill a single rectangular area in a given drawable, use *XFillRectangle*.

```
XFillRectangle(display, d, gc, x, y,
        width, height)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      unsigned int width, height;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x
y   Specify the x and y coordinates, which are relative to the origin of the
                   drawable and specify the upper-left corner of the rectangle.

width
height   Specify the width and height, which are the dimensions of the rectangle to be
                   filled.

To fill multiple rectangular areas in a given drawable, use *XFillRectangles*.

```
XFillRectangles(display, d, gc, rectangles, nrectangles)
      Display *display;
      Drawable d;
      GC gc;
      XRectangle *rectangles;
      int nrectangles;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

rectangles   Specifies an array of rectangles.

nrectangles   Specifies the number of rectangles in the array.

The *XFillRectangle* and *XFillRectangles* functions fill the specified rectangle or rectangles as if a
four-point *FillPolygon* protocol request were specified for each rectangle:

```
[x,y] [x+width,y] [x+width,y+height] [x,y+height]
```

Each function uses the x and y coordinates, width and height dimensions, and GC you specify.

XFillRectangles fills the rectangles in the order listed in the array. For any given rectangle, *XFillRectangle* and *XFillRectangles* do not draw a pixel more than once. If rectangles intersect, the intersecting pixels are drawn multiple times.

Both functions use these GC components: function, plane-mask, fill-style, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin and tile-stipple-y-origin.

XFillRectangle and *XFillRectangles* can generate *BadDrawable*, *BadGC* and *BadMatch* errors.

9.4.2   Filling a Single Polygon

To fill a polygon area in a given drawable, use *XFillPolygon*.

```
XFillPolygon(display, d, gc, points, npoints,
        shape, mode)
    Display *display;
    Drawable d;
    GC gc;
    XPoint *points;
    int npoints;
    int shape;
    int mode;
```

display	Specifies the connection to the X server.
d	Specifies the drawable.
gc	Specifies the GC.
points	Specifies an array of points.
npoints	Specifies the number of points in the array.
shape	Specifies a shape that helps the server to improve performance. You can pass *Complex*, *Convex* or *Nonconvex*.
mode	Specifies the coordinate mode. You can pass *CoordModeOrigin* or *CoordModePrevious*.

XFillPolygon fills the region closed by the specified path. The path is closed automatically if the last point in the list does not coincide with the first point. *XFillPolygon* does not draw a pixel of the region more than once. *CoordModeOrigin* treats all coordinates as relative to the origin, and *CoordModePrevious* treats all coordinates after the first as relative to the previous point.

Depending on the specified shape, the following occurs:

- If shape is *Complex*, the path may self-intersect. Note that contiguous coincident points in the path are not treated as self-intersection.

- If shape is *Convex*, for every pair of points inside the polygon, the line segment connecting them does not intersect the path. If known by the client, specifying *Convex* can improve performance. If you specify *Convex* for a path that is not convex, the graphics results are undefined.

- If shape is *Nonconvex*, the path does not self-intersect, but the shape is not wholly convex. If known by the client, specifying *Nonconvex* instead of *Complex* may improve performance. If you specify *Nonconvex* for a self-intersecting path, the graphics results are undefined.

The fill-rule of the GC controls the filling behaviour of self-intersecting polygons.

This function uses these GC components: function, plane-mask, fill-style, fill-rule, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. It also uses these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin and tile-stipple-y-origin.

XFillPolygon can generate *BadDrawable*, *BadGC*, *BadMatch* and *BadValue* errors.

9.4.3   Filling Single and Multiple Arcs

To fill a single arc in a given drawable, use *XFillArc*.

```
XFillArc(display, d, gc,  x, y, width,
        height, angle1, angle2)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      unsigned int width, height;
      int angle1, angle2;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x
y   Specify the x and y coordinates, which are relative to the origin of the drawable and specify the upper-left corner of the bounding rectangle.

width
height   Specify the width and height, which are the major and minor axes of the arc.

angle1   Specifies the start of the arc relative to the three-o'clock position from the center, in units of degrees * 64.

angle2   Specifies the path and extent of the arc relative to the start of the arc, in units of degrees * 64.

To fill multiple arcs in a given drawable, use *XFillArcs*.

```
XFillArcs(display, d, gc, arcs, narcs)
      Display *display;
      Drawable d;
      GC gc;
      XArc *arcs;
      int narcs;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

arcs   Specifies an array of arcs.

narcs   Specifies the number of arcs in the array.

For each arc, *XFillArc* or *XFillArcs* fills the region closed by the infinitely thin path described by the specified arc and, depending on the arc-mode specified in the GC, one or two line segments. For *ArcChord*, the single line segment joining the endpoints of the arc is used. For *ArcPieSlice*, the two line segments joining the endpoints of the arc with the center point are used. *XFillArcs* fills

the arcs in the order listed in the array. For any given arc, *XFillArc* and *XFillArcs* do not draw a pixel more than once. If regions intersect, the intersecting pixels are drawn multiple times.

Both functions use these GC components: function, plane-mask, fill-style, arc-mode, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin and tile-stipple-y-origin.

XFillArc and *XFillArcs* can generate *BadDrawable*, *BadGC* and *BadMatch* errors.

9.5   Font Metrics

A font is a graphical description of a set of characters that are used to increase efficiency whenever a set of small, similar sized patterns are repeatedly used.

This section discusses how to:

- load and free fonts

- obtain and free font names

- compute character string sizes

- return logical extents

- query character string sizes.

The X server loads fonts whenever a program requests a new font. The server can cache fonts for quick lookup. Fonts are global across all screens in a server. Several levels are possible when dealing with fonts. Most applications simply use *XLoadQueryFont* to load a font and query the font metrics.

Characters in fonts are regarded as masks. Except for image text requests, the only pixels modified are those in which bits are set to 1 in the character. This means that it makes sense to draw text using stipples or tiles (for example, many menus gray-out unusable entries).

The *XFontStruct* structure contains all of the information for the font and consists of the font-specific information as well as a pointer to an array of *XCharStruct* structures for the characters contained in the font. The *XFontStruct*, *XFontProp* and *XCharStruct* structures contain:

```
typedef struct {
      short lbearing;              /* origin to left edge of raster */
      short rbearing;              /* origin to right edge of raster */
      short width;                 /* advance to next char's origin */
      short ascent;                /* baseline to top edge of raster */
      short descent;               /* baseline to bottom edge of
                                      raster */
      unsigned short attributes; /* per char flags (not predefined) */
} XCharStruct;

typedef struct {
      Atom      name;
      unsigned long card32;
} XFontProp;

typedef struct {    /* normal 16 bit characters are two bytes */
      unsigned char byte1;
      unsigned char byte2;
} XChar2b;
```

```
typedef struct {
      XExtData *ext_data;        /* hook for extension to hang
                                    data */
      Font fid;                  /* Font id for this font */
      unsigned direction;        /* hint about the direction font
                                    is painted */
      unsigned min_char_or_byte2; /* first character */
      unsigned max_char_or_byte2; /* last character */
      unsigned min_byte1;        /* first row that exists */
      unsigned max_byte1;        /* last row that exists */
      Bool all_chars_exist;      /* flag if all characters have
                                    non-zero size */
      unsigned default_char;     /* char to print for undefined
                                    character */
      int n_properties;          /* how many properties there are */
      XFontProp *properties;     /* pointer to array of additional
                                    properties */
      XCharStruct min_bounds;    /* minimum bounds over all existing
                                    char */
      XCharStruct max_bounds;    /* maximum bounds over all existing
                                    char */
      XCharStruct *per_char;     /* first_char to last_char
                                    information */
      int ascent;                /* logical extent above baseline
                                    for spacing */
      int descent;               /* logical decent below baseline
                                    for spacing */
   } XFontStruct;
```

X supports single byte/character, two bytes/character matrix and 16-bit character text operations. Note that any of these forms can be used with a font, but a single byte/character text request can only specify a single byte (that is, the first row of a 2-byte font). You should view 2-byte fonts as a two-dimensional matrix of defined characters: byte1 specifies the range of defined rows and byte2 defines the range of defined columns of the font. Single byte/character fonts have one row defined, and the byte2 range specified in the structure defines a range of characters.

The bounding box of a character is defined by the *XCharStruct* of that character. When characters are absent from a font, the default_char is used. When fonts have all characters of the same size, only the information in the *XFontStruct* min and max bounds are used.

The members of the *XFontStruct* have the following semantics:

• The direction member can be either *FontLeftToRight* or *FontRightToLeft*. It is just a hint as to whether most *XCharStruct* elements have a positive (*FontLeftToRight*) or a negative (*FontRightToLeft*) character width metric. The core protocol defines no support for vertical text.

• If the min_byte1 and max_byte1 members are both zero, min_char_or_byte2 specifies the linear character index corresponding to the first element of the per_char array, and max_char_or_byte2 specifies the linear character index of the last element.

If either min_byte1 or max_byte1 are non-zero, both min_char_or_byte2 and max_char_or_byte2 are less than 256, and the 2-byte character index values corresponding to the per_char array element N (counting from 0) are:

```
byte1 = N/D + min_byte1
byte2 = N\D + min_char_or_byte2
```

where:

```
D = max_char_or_byte2 − min_char_or_byte2 + 1
/ = integer division
\ = integer modulus
```

- If the per_char pointer is NULL, all glyphs between the first and last character indexes inclusive have the same information, as given by both min_bounds and max_bounds.

- If all_chars_exist is *True*, all characters in the per_char array have non-zero bounding boxes.

- The default_char member specifies the character that will be used when an undefined or nonexistent character is printed. The default_char is a 16-bit character (not a 2-byte character). For a font using 2-byte matrix format, the default_char has byte1 in the most-significant byte and byte2 in the least-significant byte. If the default_char itself specifies an undefined or nonexistent character, no printing is performed for an undefined or nonexistent character.

- The min_bounds and max_bounds members contain the most extreme values of each individual *XCharStruct* component over all elements of this array (and ignore nonexistent characters). The bounding box of the font (the smallest rectangle enclosing the shape obtained by superimposing all of the characters at the same origin [x,y]) has its upper-left coordinate at:

```
[x + min_bounds.lbearing, y − max_bounds.ascent]
```

Its width is:

```
max_bounds.rbearing − min_bounds.lbearing
```

Its height is:

```
max_bounds.ascent + max_bounds.descent
```

- The ascent member is the logical extent of the font above the baseline that is used for determining line spacing. Specific characters may extend beyond this.

- The descent member is the logical extent of the font at or below the baseline that is used for determining line spacing. Specific characters may extend beyond this.

- If the baseline is at Y-coordinate y, the logical extent of the font is inclusive between the Y-coordinate values (y − font.ascent) and (y + font.descent − 1). Typically, the minimum interline spacing between rows of text is given by ascent + descent.

For a character origin at [x,y], the bounding box of a character (that is, the smallest rectangle that encloses the character's shape) described in terms of *XCharStruct* components is a rectangle with its upper-left corner at:

```
[x + lbearing, y − ascent]
```

Its width is:

```
rbearing − lbearing
```

Its height is:

```
ascent + descent
```

The origin for the next character is defined to be:

```
[x + width, y]
```

The lbearing member defines the extent of the left edge of the character ink from the origin. The rbearing member defines the extent of the right edge of the character ink from the origin. The ascent member defines the extent of the top edge of the character ink from the origin. The descent member defines the extent of the bottom edge of the character ink from the origin. The width member defines the logical width of the character.

Note that the baseline (the y position of the character origin) is logically viewed as being the scanline just below non-descending characters. When descent is zero, only pixels with Y-coordinates less than y are drawn, and the origin is logically viewed as being coincident with the left edge of a non-kerned character. When lbearing is zero, no pixels with X-coordinate less than x are drawn. Any of the *XCharStruct* metric members could be negative. If the width is negative, the next character will be placed to the left of the current origin.

The X protocol does not define the interpretation of the attributes member in the *XCharStruct* structure. A nonexistent character is represented with all members of its *XCharStruct* set to zero.

A font is not guaranteed to have any properties. The interpretation of the property value (for example, long or unsigned long) must be derived from *a priori* knowledge of the property. A basic set of font properties is specified in the **XLFD** specification.

9.5.1   Loading and Freeing Fonts

Xlib provides functions that you can use to load fonts, get font information, unload fonts and free font information. A few font functions use a *GContext* resource ID or a font ID interchangeably.

To load a given font, use *XLoadFont*.

```
Font XLoadFont(display, name)
      Display *display;
      char *name;
```

display   Specifies the connection to the X server.

name   Specifies the name of the font, which is a null-terminated string.

The *XLoadFont* function loads the specified font and returns its associated font ID. If the font name is not in the Host Portable Character Encoding the result is implementation-dependent. Use of upper case or lower case does not matter. The interpretation of characters ''?'' and ''*'' in the name is not defined by the core protocol but is reserved for future definition. A structured format for font names is specified in the **XLFD** specification. If *XLoadFont* was unsuccessful at loading the specified font, a *BadName* error results. Fonts are not associated with a particular screen and can be stored as a component of any GC. When the font is no longer needed, call *XUnloadFont*.

XLoadFont can generate *BadAlloc* and *BadName* errors.

To return information about an available font, use *XQueryFont*.

```
XFontStruct *XQueryFont(display, font_ID)
      Display *display;
      XID font_ID;
```

display   Specifies the connection to the X server.

font_ID   Specifies the font ID or the *GContext* ID.

The *XQueryFont* function returns a pointer to the *XFontStruct* structure, which contains information associated with the font. You can query a font or the font stored in a GC. The font ID stored in the *XFontStruct* structure will be the *GContext* ID, and you need to be careful when using this ID in other functions (see *XGContextFromGC*). If the font does not exist, *XQueryFont* returns NULL. To free this data, use *XFreeFontInfo*.

To perform a *XLoadFont* and *XQueryFont* in a single operation, use *XLoadQueryFont*.

```
XFontStruct *XLoadQueryFont(display, name)
      Display *display;
      char *name;
```

display   Specifies the connection to the X server.

name   Specifies the name of the font, which is a null-terminated string.

The *XLoadQueryFont* function provides the most common way for accessing a font. *XLoadQueryFont* both opens (loads) the specified font and returns a pointer to the appropriate *XFontStruct* structure. If the font name is not in the Host Portable Character Encoding the result is implementation-dependent. If the font does not exist, *XLoadQueryFont* returns NULL.

XLoadQueryFont can generate a *BadAlloc* error.

To unload the font and free the storage used by the font structure that was allocated by *XQueryFont* or *XLoadQueryFont*, use *XFreeFont*.

```
XFreeFont(display, font_struct)
      Display *display;
      XFontStruct *font_struct;
```

display   Specifies the connection to the X server.

font_struct   Specifies the storage associated with the font.

The *XFreeFont* function deletes the association between the font resource ID and the specified font and frees the *XFontStruct* structure. The font itself will be freed when no other resource references it. The data and the font should not be referenced again.

XFreeFont can generate a *BadFont* error.

To return a given font property, use *XGetFontProperty*.

```
Bool XGetFontProperty(font_struct, atom, value_return)
      XFontStruct *font_struct;
      Atom atom;
      unsigned long *value_return;
```

font_struct   Specifies the storage associated with the font.

atom   Specifies the atom for the property name you want returned.

value_return   Returns the value of the font property.

Given the atom for that property, the *XGetFontProperty* function returns the value of the specified font property. *XGetFontProperty* also returns *False* if the property was not defined or *True* if it was defined. A set of predefined atoms exists for font properties, which can be found in **<X11/Xatom.h>**. This set contains the standard properties associated with a font. Although it is not guaranteed, it is likely that the predefined font properties will be present.

To unload a font that was loaded by *XLoadFont*, use *XUnloadFont*.

```
XUnloadFont(display, font)
      Display *display;
      Font font;
```

display   Specifies the connection to the X server.

font   Specifies the font.

The *XUnloadFont* function deletes the association between the font resource ID and the specified font. The font itself will be freed when no other resource references it. The font should not be referenced again.

XUnloadFont can generate a *BadFont* error.

9.5.2   Obtaining and Freeing Font Names and Information

You obtain font names and information by matching a wildcard specification when querying a font type for a list of available sizes and so on.

To return a list of the available font names, use *XListFonts*.

```
char **XListFonts(display, pattern, maxnames,
      actual_count_return)
      Display *display;
      char *pattern;
      int maxnames;
      int *actual_count_return;
```

display   Specifies the connection to the X server.

pattern   Specifies the null-terminated pattern string that can contain wildcard characters.

maxnames   Specifies the maximum number of names to be returned.

actual_count_return Returns the actual number of font names.

The *XListFonts* function returns an array of available font names (as controlled by the font search path; see *XSetFontPath*) that match the string you passed to the pattern argument. The pattern string can contain any characters, but each asterisk (*) is a wildcard for any number of characters, and each question mark (?) is a wildcard for a single character. If the pattern string is not in the Host Portable Character Encoding the result is implementation-dependent. Use of upper case or lower case does not matter. Each returned string is null-terminated. If the data returned by the server is in the Latin Portable Character Encoding, then the returned strings are in the Host Portable Character Encoding. Otherwise, the result is implementation-dependent. If there are no matching font names, *XListFonts* returns NULL. The client should call *XFreeFontNames* when finished with the result to free the memory.

To free a font name array, use *XFreeFontNames*.

```
XFreeFontNames(list)
      char *list[];
```

list   Specifies the array of strings you want to free.

The *XFreeFontNames* function frees the array and strings returned by *XListFonts* or *XListFontsWithInfo*.

To obtain the names and information about available fonts, use *XListFontsWithInfo*.

```
      char **XListFontsWithInfo(display, pattern, maxnames,
            count_return, info_return)
         Display *display;
         char *pattern;
         int maxnames;
         int *count_return;
         XFontStruct **info_return;
```

display   Specifies the connection to the X server.

pattern   Specifies the null-terminated pattern string that can contain wildcard
                   characters.

maxnames   Specifies the maximum number of names to be returned.

count_return   Returns the actual number of matched font names.

info_return   Returns the font information.

The *XListFontsWithInfo* function returns a list of font names that match the specified pattern and
their associated font information. The list of names is limited to size specified by maxnames.
The information returned for each font is identical to what *XLoadQueryFont* would return except
that the per-character metrics are not returned. The pattern string can contain any characters,
but each asterisk (*) is a wildcard for any number of characters, and each question mark (?) is a
wildcard for a single character. If the pattern string is not in the Host Portable Character
Encoding the result is implementation-dependent. Use of upper case or lower case does not
matter. Each returned string is null-terminated. If the data returned by the server is in the Latin
Portable Character Encoding, then the returned strings are in the Host Portable Character
Encoding. Otherwise, the result is implementation-dependent. If there are no matching font
names, *XListFontsWithInfo* returns NULL.

To free only the allocated name array, the client should call *XFreeFontNames*. To free both the
name array and the font information array, or to free just the font information array, the client
should call *XFreeFontInfo*.

To free the the font information array, use *XFreeFontInfo*.

```
      XFreeFontInfo(names, free_info, actual_count)
         char **names;
         XFontStruct *free_info;
         int actual_count;
```

names   Specifies the list of font names returned by *XListFontsWithInfo*.

free_info   Specifies the font information returned by *XListFontsWithInfo*.

actual_count   Specifies the actual number of matched font names returned by
                   XListFontsWithInfo.

The *XFreeFontInfo* function frees the the font information array. To free an *XFontStruct* structure
without closing the font, call *XFreeFontInfo* with the names argument specified as NULL.

9.5.3   Computing Character String Sizes

Xlib provides functions that you can use to compute the width, the logical extents, and the server information about 8-bit and 2-byte text strings. The width is computed by adding the character widths of all the characters. It does not matter if the font is an 8-bit or 2-byte font. These functions return the sum of the character metrics, in pixels.

To determine the width of an 8-bit character string, use *XTextWidth*.

```
int XTextWidth(font_struct, string, count)
      XFontStruct *font_struct;
      char *string;
      int count;
```

font_struct   Specifies the font used for the width computation.

string   Specifies the character string.

count   Specifies the character count in the specified string.

To determine the width of a 2-byte character string, use *XTextWidth16*.

```
int XTextWidth16(font_struct, string, count)
      XFontStruct *font_struct;
      XChar2b *string;
      int count;
```

font_struct   Specifies the font used for the width computation.

string   Specifies the character string.

count   Specifies the character count in the specified string.

9.5.4   Computing Logical Extents

To compute the bounding box of an 8-bit character string in a given font, use *XTextExtents*.

```
XTextExtents(font_struct, string, nchars,
        direction_return, font_ascent_return,
        font_descent_return, overall_return)
      XFontStruct *font_struct;
      char *string;
      int nchars;
      int *direction_return;
      int *font_ascent_return, *font_descent_return;
      XCharStruct *overall_return;
```

font_struct   Specifies the *XFontStruct* structure.

string   Specifies the character string.

nchars   Specifies the number of characters in the character string.

direction_return   Returns the value of the direction hint (*FontLeftToRight* or *FontRightToLeft*).

font_ascent_return   Returns the font ascent.

font_descent_return
                       Returns the font descent.

overall_return   Returns the overall size in the specified *XCharStruct* structure.

To compute the bounding box of a 2-byte character string in a given font, use *XTextExtents16*.

```
XTextExtents16(font_struct, string, nchars,
        direction_return, font_ascent_return,
        font_descent_return, overall_return)
    XFontStruct *font_struct;
    XChar2b *string;
    int nchars;
    int *direction_return;
    int *font_ascent_return, *font_descent_return;
    XCharStruct *overall_return;
```

font_struct   Specifies the *XFontStruct* structure.

string   Specifies the character string.

nchars   Specifies the number of characters in the character string.

direction_return   Returns the value of the direction hint (*FontLeftToRight* or *FontRightToLeft*).

font_ascent_return Returns the font ascent.

font_descent_return
               Returns the font descent.

overall_return   Returns the overall size in the specified *XCharStruct* structure.

The *XTextExtents* and *XTextExtents16* functions perform the size computation locally and, thereby, avoid the round-trip overhead of *XQueryTextExtents* and *XQueryTextExtents16*. Both functions return an *XCharStruct* structure, whose members are set to the values as follows.

The ascent member is set to the maximum of the ascent metrics of all characters in the string. The descent member is set to the maximum of the descent metrics. The width member is set to the sum of the character-width metrics of all characters in the string. For each character in the string, let W be the sum of the character-width metrics of all characters preceding it in the string. Let L be the left-side-bearing metric of the character plus W. Let R be the right-side-bearing metric of the character plus W. The lbearing member is set to the minimum L of all characters in the string. The rbearing member is set to the maximum R.

For fonts defined with linear indexing rather than 2-byte matrix indexing, each *XChar2b* structure is interpreted as a 16-bit number with byte1 as the most-significant byte. If the font has no defined default character, undefined characters in the string are taken to have all zero metrics.

9.5.5   Querying Character String Sizes

To query the server for the bounding box of an 8-bit character string in a given font, use *XQueryTextExtents*.

```
XQueryTextExtents(display, font_ID, string, nchars,
        direction_return, font_ascent_return,
        font_descent_return, overall_return)
    Display *display;
    XID font_ID;
    char *string;
    int nchars;
    int *direction_return;
    int *font_ascent_return, *font_descent_return;
    XCharStruct *overall_return;
```

display   Specifies the connection to the X server.

font_ID   Specifies either the font ID or the *GContext* ID that contains the font.

string   Specifies the character string.

nchars   Specifies the number of characters in the character string.

direction_return   Returns the value of the direction hint (*FontLeftToRight* or *FontRightToLeft*).

font_ascent_return Returns the font ascent.

font_descent_return

                         Returns the font descent.

overall_return   Returns the overall size in the specified *XCharStruct* structure.

To query the server for the bounding box of a 2-byte character string in a given font, use *XQueryTextExtents16*.

```
XQueryTextExtents16(display, font_ID, string, nchars,
        direction_return, font_ascent_return,
        font_descent_return, overall_return)
    Display *display;
    XID font_ID;
    XChar2b *string;
    int nchars;
    int *direction_return;
    int *font_ascent_return, *font_descent_return;
    XCharStruct *overall_return;
```

display   Specifies the connection to the X server.

font_ID   Specifies either the font ID or the *GContext* ID that contains the font.

string   Specifies the character string.

nchars   Specifies the number of characters in the character string.

direction_return   Returns the value of the direction hint (*FontLeftToRight* or *FontRightToLeft*).

font_ascent_return Returns the font ascent.

font_descent_return

                         Returns the font descent.

overall_return   Returns the overall size in the specified *XCharStruct* structure.

The *XQueryTextExtents* and *XQueryTextExtents16* functions return the bounding box of the specified 8-bit and 16-bit character string in the specified font or the font contained in the specified GC. These functions query the X server and, therefore, suffer the round-trip overhead that is avoided by *XTextExtents* and *XTextExtents16*. Both functions return a *XCharStruct* structure, whose members are set to the values as follows.

The ascent member is set to the maximum of the ascent metrics of all characters in the string. The descent member is set to the maximum of the descent metrics. The width member is set to the sum of the character-width metrics of all characters in the string. For each character in the string, let W be the sum of the character-width metrics of all characters preceding it in the string. Let L be the left-side-bearing metric of the character plus W. Let R be the right-side-bearing metric of the character plus W. The lbearing member is set to the minimum L of all characters in the string. The rbearing member is set to the maximum R.

For fonts defined with linear indexing rather than 2-byte matrix indexing, each *XChar2b* structure is interpreted as a 16-bit number with byte1 as the most-significant byte. If the font has no defined default character, undefined characters in the string are taken to have all zero metrics.

Characters with all zero metrics are ignored. If the font has no defined default_char, the undefined characters in the string are also ignored.

XQueryTextExtents and *XQueryTextExtents16* can generate *BadFont* and *BadGC* errors.

9.6   Drawing Text

This section discusses how to draw:

- complex text

- text characters

- image text characters.

The fundamental text functions *XDrawText* and *XDrawText16* use the following structures.

```
typedef struct {
      char *chars;     /* pointer to string */
      int nchars;      /* number of characters */
      int delta;       /* delta between strings */
      Font font;       /* Font to print it in, None don't change */
} XTextItem;

typedef struct {
      XChar2b *chars; /* pointer to two-byte characters */
      int nchars;      /* number of characters */
      int delta;       /* delta between strings */
      Font font;       /* font to print it in, None don't change */
} XTextItem16;
```

If the font member is not *None*, the font is changed before printing and also is stored in the GC. If an error was generated during text drawing, the previous items may have been drawn. The baseline of the characters are drawn starting at the x and y coordinates that you pass in the text drawing functions.

For example, consider the background rectangle drawn by *XDrawImageString*. If you want the upper-left corner of the background rectangle to be at pixel coordinate (x,y), pass the (x,y + ascent) as the baseline origin coordinates to the text functions. The ascent is the font ascent, as given in the *XFontStruct* structure. If you want the lower-left corner of the background rectangle to be at pixel coordinate (x,y), pass the (x,y − descent + 1) as the baseline origin coordinates to the text functions. The descent is the font descent, as given in the *XFontStruct* structure.

9.6.1   Drawing Complex Text

To draw 8-bit characters in a given drawable, use *XDrawText*.

```
XDrawText(display, d, gc, x, y, items, nitems)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XTextItem *items;
      int nitems;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x

y   Specify the x and y coordinates, which are relative to the origin of the specified drawable and define the origin of the first character.

items   Specifies an array of text items.

nitems   Specifies the number of text items in the array.

To draw 2-byte characters in a given drawable, use *XDrawText16*.

```
XDrawText16(display, d, gc, x, y, items, nitems)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XTextItem16 *items;
      int nitems;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x

y   Specify the x and y coordinates, which are relative to the origin of the specified drawable and define the origin of the first character.

items   Specifies an array of text items.

nitems   Specifies the number of text items in the array.

The *XDrawText16* function is similar to *XDrawText* except that it uses 2-byte or 16-bit characters. Both functions allow complex spacing and font shifts between counted strings.

Each text item is processed in turn. A font member other than *None* in an item causes the font to be stored in the GC and used for subsequent text. A text element delta specifies an additional change in the position along the x axis before the string is drawn. The delta is always added to the character origin and is not dependent on any characteristics of the font. Each character image, as defined by the font in the GC, is treated as an additional mask for a fill operation on the drawable. The drawable is modified only where the font character has a bit set to 1. If a text item generates a *BadFont* error, the previous text items may have been drawn.

For fonts defined with linear indexing rather than 2-byte matrix indexing, each *XChar2b* structure is interpreted as a 16-bit number with byte1 as the most-significant byte.

Both functions use these GC components: function, plane-mask, fill-style, font, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin and tile-stipple-y-origin.

XDrawText and *XDrawText16* can generate *BadDrawable*, *BadFont*, *BadGC* and *BadMatch* errors.

9.6.2   Drawing Text Characters

To draw 8-bit characters in a given drawable, use *XDrawString*.

```
XDrawString(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      char *string;
      int length;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x
y   Specify the x and y coordinates, which are relative to the origin of the
                     specified drawable and define the origin of the first character.

string   Specifies the character string.

length   Specifies the number of characters in the string argument.

To draw 2-byte characters in a given drawable, use *XDrawString16*.

```
XDrawString16(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XChar2b *string;
      int length;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x
y   Specify the x and y coordinates, which are relative to the origin of the
                     specified drawable and define the origin of the first character.

string   Specifies the character string.

length   Specifies the number of characters in the string argument.

Each character image, as defined by the font in the GC, is treated as an additional mask for a fill operation on the drawable. The drawable is modified only where the font character has a bit set to 1. For fonts defined with 2-byte matrix indexing and used with *XDrawString16*, each byte is used as a byte2 with a byte1 of zero.

Both functions use these GC components: function, plane-mask, fill-style, font, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin and tile-stipple-y-origin.

XDrawString and *XDrawString16* can generate *BadDrawable*, *BadGC* and *BadMatch* errors.

9.6.3   Drawing Image Text Characters

Some applications, in particular terminal emulators, need to print image text in which both the foreground and background bits of each character are painted. This prevents annoying flicker on many displays.

To draw 8-bit image text characters in a given drawable, use *XDrawImageString*.

```
XDrawImageString(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      char *string;
      int length;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x

y   Specify the x and y coordinates, which are relative to the origin of the
                   specified drawable and define the origin of the first character.

string   Specifies the character string.

length   Specifies the number of characters in the string argument.

To draw 2-byte image text characters in a given drawable, use *XDrawImageString16*.

```
XDrawImageString16(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XChar2b *string;
      int length;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x

y   Specify the x and y coordinates, which are relative to the origin of the
                   specified drawable and define the origin of the first character.

string   Specifies the character string.

length   Specifies the number of characters in the string argument.

The *XDrawImageString16* function is similar to *XDrawImageString* except that it uses 2-byte or
16-bit characters. Both functions also use both the foreground and background pixels of the GC
in the destination.

The effect is first to fill a destination rectangle with the background pixel defined in the GC and
then to paint the text with the foreground pixel. The upper-left corner of the filled rectangle is
at:

```
[x, y − font-ascent]
```

The width is:

```
overall-width
```

The height is:

```
font-ascent + font-descent
```

The overall-width, font-ascent and font-descent are as would be returned by *XQueryTextExtents* using gc and string. The function and fill-style defined in the GC are ignored for these functions. The effective function is *GXcopy*, and the effective fill-style is *FillSolid*.

For fonts defined with 2-byte matrix indexing and used with *XDrawImageString*, each byte is used as a byte2 with a byte1 of zero.

Both functions use these GC components: plane-mask, foreground, background, font, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask.

XDrawImageString and *XDrawImageString16* can generate *BadDrawable*, *BadGC* and *BadMatch* errors.

9.7   Transferring Images between Client and Server

Xlib provides functions that you can use to transfer images between a client and the server. Because the server may require diverse data formats, Xlib provides an image object that fully describes the data in memory and that provides for basic operations on that data. You should reference the data through the image object rather than referencing the data directly. However, some implementations of the Xlib library may efficiently deal with frequently used data formats by replacing functions in the procedure vector with special case functions. Supported operations include destroying the image, getting a pixel, storing a pixel, extracting a subimage of an image, and adding a constant to an image (see Section 17.5).

All the image manipulation functions discussed in this section make use of the *XImage* data structure, which describes an image as it exists in the client's memory.

```
typedef struct _XImage {
      int width, height;          /* size of image */
      int xoffset;                /* number of pixels offset in X
                                      direction */
      int format;                 /* XYBitmap, XYPixmap, ZPixmap */
      char *data;                 /* pointer to image data */
      int byte_order;             /* data byte order, LSBFirst,
                                      MSBFirst */
      int bitmap_unit;            /* quant. of scanline 8, 16, 32 */
      int bitmap_bit_order;       /* LSBFirst, MSBFirst */
      int bitmap_pad;             /* 8, 16, 32 either XY or ZPixmap */
      int depth;                  /* depth of image */
      int bytes_per_line;         /* accelerator to next scanline */
      int bits_per_pixel;         /* bits per pixel (ZPixmap) */
      unsigned long red_mask;     /* bits in z arrangement */
      unsigned long green_mask;
      unsigned long blue_mask;
      XPointer obdata;            /* hook for the object routines to
                                      hang on */
      struct funcs {              /* image manipulation routines */
            struct _XImage *(*create_image)();
            int (*destroy_image)();
            unsigned long (*get_pixel)();
            int (*put_pixel)();
            struct _XImage *(*sub_image)();
            int (*add_pixel)();
      } f;
} XImage;
```

You may request that some of the members (for example, height, width and xoffset) be changed when the image is sent to the server. That is, you may send a subset of the image. Other members (for example, byte_order, bitmap_unit, and so on) are characteristics of both the image and the server. If these members differ between the image and the server, *XPutImage* makes the appropriate conversions. The first byte of the first scanline of plane n is located at the address (data + (n * height * bytes_per_line)).

To combine an image in memory with a rectangle of a drawable on the display, use *XPutImage*.

```
XPutImage(display, d, gc, image, src_x,
        src_y, dest_x, dest_y, width, height)
      Display *display;
      Drawable d;
      GC gc;
      XImage *image;
      int src_x, src_y;
      int dest_x, dest_y;
      unsigned int width, height;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

image   Specifies the image you want combined with the rectangle.

src_x   Specifies the offset in X from the left edge of the image defined by the *XImage*
                   data structure.

src_y   Specifies the offset in Y from the top edge of the image defined by the *XImage*
                   data structure.

dest_x
dest_y   Specify the x and y coordinates, which are relative to the origin of the
                   drawable and are the coordinates of the subimage.

width
height   Specify the width and height of the subimage, which define the dimensions of
                   the rectangle.

The *XPutImage* function combines an image in memory with a rectangle of the specified
drawable. If *XYBitmap* format is used, the depth of the image must be one, or a *BadMatch* error
results. The foreground pixel in the GC defines the source for the one bits in the image, and the
background pixel defines the source for the zero bits. For *XYPixmap* and *ZPixmap*, the depth of
the image must match the depth of the drawable, or a *BadMatch* error results. The section of the
image defined by the src_x, src_y, width and height arguments is drawn on the specified part of
the drawable.

This function uses these GC components: function, plane-mask, subwindow-mode, clip-x-
origin, clip-y-origin and clip-mask. It also uses these GC mode-dependent components:
foreground and background.

XPutImage can generate *BadDrawable*, *BadGC*, *BadMatch* and *BadValue* errors.

To return the contents of a rectangle in a given drawable on the display, use *XGetImage*. This
function specifically supports rudimentary screen dumps.

```
XImage *XGetImage(display, d, x, y, width,
        height, plane_mask, format)
      Display *display;
      Drawable d;
      int x, y;
      unsigned int width, height;
      unsigned long plane_mask;
      int format;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

x
y   Specify the x and y coordinates, which are relative to the origin of the drawable and define the upper-left corner of the rectangle.

width
height   Specify the width and height of the subimage, which define the dimensions of the rectangle.

plane_mask   Specifies the plane mask.

format   Specifies the format for the image. You can pass *XYPixmap* or *ZPixmap*.

The *XGetImage* function returns a pointer to an *XImage* structure. This structure provides you with the contents of the specified rectangle of the drawable in the format you specify. If the format argument is *XYPixmap*, the image contains only the bit planes you passed to the plane_mask argument. If the plane_mask argument only requests a subset of the planes of the display, the depth of the returned image will be the number of planes requested. If the format argument is *ZPixmap*, *XGetImage* returns as zero the bits in all planes not specified in the plane_mask argument. The function performs no range checking on the values in plane_mask and ignores extraneous bits.

XGetImage returns the depth of the image to the depth member of the *XImage* structure. The depth of the image is as specified when the drawable was created, except when getting a subset of the planes in *XYPixmap* format, when the depth is given by the number of bits set to 1 in plane_mask.

If the drawable is a pixmap, the given rectangle must be wholly contained within the pixmap, or a *BadMatch* error results. If the drawable is a window, the window must be viewable, and it must be the case that if there were no inferiors or overlapping windows, the specified rectangle of the window would be fully visible on the screen and wholly contained within the outside edges of the window, or a *BadMatch* error results. Note that the borders of the window can be included and read with this request. If the window has backing-store, the backing-store contents are returned for regions of the window that are obscured by non-inferior windows. If the window does not have backing-store, the returned contents of such obscured regions are undefined. The returned contents of visible regions of inferiors of a different depth than the specified window's depth are also undefined. The pointer cursor image is not included in the returned contents. If a problem occurs, *XGetImage* returns NULL.

XGetImage can generate *BadDrawable*, *BadMatch* and *BadValue* errors.

To copy the contents of a rectangle on the display to a location within a preexisting image structure, use *XGetSubImage*.

```
XImage *XGetSubImage(display, d, x, y, width,
        height, plane_mask, format, dest_image,
        dest_x, dest_y)
    Display *display;
    Drawable d;
    int x, y;
    unsigned int width, height;
    unsigned long plane_mask;
    int format;
    XImage *dest_image;
    int dest_x, dest_y;
```

display　　　　　　Specifies the connection to the X server.

d　　　　　　　　Specifies the drawable.

x

y　　　　　　　　Specify the x and y coordinates, which are relative to the origin of the drawable and define the upper-left corner of the rectangle.

width

height　　　　　　Specify the width and height of the subimage, which define the dimensions of the rectangle.

plane_mask　　　　Specifies the plane mask.

format　　　　　　Specifies the format for the image. You can pass *XYPixmap* or *ZPixmap*.

dest_image　　　　Specify the destination image.

dest_x

dest_y　　　　　　Specify the x and y coordinates, which are relative to the origin of the destination rectangle, specify its upper-left corner, and determine where the subimage is placed in the destination image.

The *XGetSubImage* function updates dest_image with the specified subimage in the same manner as *XGetImage*. If the format argument is *XYPixmap*, the image contains only the bit planes you passed to the plane_mask argument. If the format argument is *ZPixmap*, *XGetSubImage* returns as zero the bits in all planes not specified in the plane_mask argument. The function performs no range checking on the values in plane_mask and ignores extraneous bits. As a convenience, *XGetSubImage* returns a pointer to the same *XImage* structure specified by dest_image.

The depth of the destination *XImage* structure must be the same as that of the drawable. If the specified subimage does not fit at the specified location on the destination image, the right and bottom edges are clipped. If the drawable is a pixmap, the given rectangle must be wholly contained within the pixmap, or a *BadMatch* error results. If the drawable is a window, the window must be viewable, and it must be the case that if there were no inferiors or overlapping windows, the specified rectangle of the window would be fully visible on the screen and wholly contained within the outside edges of the window, or a *BadMatch* error results. If the window has backing-store, then the backing-store contents are returned for regions of the window that are obscured by non-inferior windows. If the window does not have backing-store, the returned contents of such obscured regions are undefined. The returned contents of visible regions of inferiors of a different depth than the specified window's depth are also undefined. If a problem occurs, *XGetSubImage* returns NULL.

XGetSubImage can generate *BadDrawable*, *BadGC*, *BadMatch* and *BadValue* errors.

Window and Session Manager Functions

Although it is difficult to categorize functions as exclusively for an application or a window manager or a session manager, the functions in this chapter are most often used by window managers and session managers. It is not expected that these functions will be used by most application programs. Xlib provides management functions to:

- change the parent of a window
- control the lifetime of a window
- manage installed colormaps
- set and retrieve the font search path
- grab the server
- kill a client
- control the screen saver
- control host access.

10.1   Changing the Parent of a Window

To change a window's parent to another window on the same screen, use *XReparentWindow*. There is no way to move a window between screens.

```
XReparentWindow(display, w, parent, x, y)
       Display *display;
       Window w;
       Window parent;
       int x, y;
```

display	Specifies the connection to the X server.
w	Specifies the window.
parent	Specifies the parent window.
x	
y	Specify the x and y coordinates of the position in the new parent window.

If the specified window is mapped, *XReparentWindow* automatically performs an *UnmapWindow* request on it, removes it from its current position in the hierarchy, and inserts it as the child of the specified parent. The window is placed in the stacking order on top with respect to sibling windows.

After reparenting the specified window, *XReparentWindow* causes the X server to generate a *ReparentNotify* event. The override_redirect member returned in this event is set to the window's corresponding attribute. Window manager clients usually should ignore this window if this member is set to *True*. Finally, if the specified window was originally mapped, the X server automatically performs a *MapWindow* request on it.

The X server performs normal exposure processing on formerly obscured windows. The X server might not generate *Expose* events for regions from the initial *UnmapWindow* request that are immediately obscured by the final *MapWindow* request. A *BadMatch* error results if:

- The new parent window is not on the same screen as the old parent window.

- The new parent window is the specified window or an inferior of the specified window.

- The new parent is *InputOnly* and the window is not.

- The specified window has a *ParentRelative* background, and the new parent window is not the same depth as the specified window.

XReparentWindow can generate *BadMatch* and *BadWindow* errors.

Stamp:XXXXXXXXXXXXXXXXXXXXXXXXX

10.2   Controlling the Lifetime of a Window

The save-set of a client is a list of other clients' windows that, if they are inferiors of one of the client's windows at connection close, should not be destroyed and should be remapped if they are unmapped. For further information about close-connection processing, see Section 3.6. To allow an application's window to survive when a window manager that has reparented a window fails, Xlib provides the save-set functions that you can use to control the longevity of subwindows that are normally destroyed when the parent is destroyed. For example, a window manager that wants to add decoration to a window by adding a frame might reparent an application's window. When the frame is destroyed, the application's window should not be destroyed but be returned to its previous place in the window hierarchy.

The X server automatically removes windows from the save-set when they are destroyed.

To add or remove a window from the client's save-set, use *XChangeSaveSet*.

```
XChangeSaveSet(display, w, change_mode)
      Display *display;
      Window w;
      int change_mode;
```

display   Specifies the connection to the X server.

w   Specifies the window that you want to add to or delete from the client's save-set.

change_mode   Specifies the mode. You can pass *SetModeInsert* or *SetModeDelete*.

Depending on the specified mode, *XChangeSaveSet* either inserts or deletes the specified window from the client's save-set. The specified window must have been created by some other client, or a *BadMatch* error results.

XChangeSaveSet can generate *BadMatch*, *BadValue* and *BadWindow* errors.

To add a window to the client's save-set, use *XAddToSaveSet*.

```
XAddToSaveSet(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window that you want to add to the client's save-set.

The *XAddToSaveSet* function adds the specified window to the client's save-set. The specified window must have been created by some other client, or a *BadMatch* error results.

XAddToSaveSet can generate *BadMatch* and *BadWindow* errors.

To remove a window from the client's save-set, use *XRemoveFromSaveSet*.

```
XRemoveFromSaveSet(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window that you want to delete from the client's save-set.

The *XRemoveFromSaveSet* function removes the specified window from the client's save-set. The specified window must have been created by some other client, or a *BadMatch* error results.

XRemoveFromSaveSet can generate *BadMatch* and *BadWindow* errors.

10.3   Managing Installed Colormaps

The X server maintains a list of installed colormaps. Windows using these colormaps are guaranteed to display with correct colors; windows using other colormaps may or may not display with correct colors. Xlib provides functions that you can use to install a colormap, uninstall a colormap, and obtain a list of installed colormaps.

At any time, there is a subset of the installed maps that is viewed as an ordered list and is called the required list. The length of the required list is at most M, where M is the minimum number of installed colormaps specified for the screen in the connection setup. The required list is maintained as follows. When a colormap is specified to *XInstallColormap*, it is added to the head of the list; the list is truncated at the tail, if necessary, to keep its length to at most M. When a colormap is specified to *XUninstallColormap* and it is in the required list, it is removed from the list. A colormap is not added to the required list when it is implicitly installed by the X server, and the X server cannot implicitly uninstall a colormap that is in the required list.

To install a colormap, use *XInstallColormap*.

```
XInstallColormap(display, colormap)
      Display *display;
      Colormap colormap;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

The *XInstallColormap* function installs the specified colormap for its associated screen. All windows associated with this colormap immediately display with true colors. You associated the windows with this colormap when you created them by calling *XCreateWindow*, *XCreateSimpleWindow*, *XChangeWindowAttributes* or *XSetWindowColormap*.

If the specified colormap is not already an installed colormap, the X server generates a *ColormapNotify* event on each window that has that colormap. In addition, for every other colormap that is installed as a result of a call to *XInstallColormap*, the X server generates a *ColormapNotify* event on each window that has that colormap.

XInstallColormap can generate a *BadColor* error.

To uninstall a colormap, use *XUninstallColormap*.

```
XUninstallColormap(display, colormap)
      Display *display;
      Colormap colormap;
```

display   Specifies the connection to the X server.

colormap   Specifies the colormap.

The *XUninstallColormap* function removes the specified colormap from the required list for its screen. As a result, the specified colormap might be uninstalled, and the X server might implicitly install or uninstall additional colormaps. Which colormaps get installed or uninstalled is server-dependent except that the required list must remain installed.

If the specified colormap becomes uninstalled, the X server generates a *ColormapNotify* event on each window that has that colormap. In addition, for every other colormap that is installed or uninstalled as a result of a call to *XUninstallColormap*, the X server generates a *ColormapNotify* event on each window that has that colormap.

XUninstallColormap can generate a *BadColor* error.

To obtain a list of the currently installed colormaps for a given screen, use
XListInstalledColormaps.

```
Colormap *XListInstalledColormaps(display, w, num_return)
      Display *display;
      Window w;
      int *num_return;
```

display   Specifies the connection to the X server.

w   Specifies the window that determines the screen.

num_return   Returns the number of currently installed colormaps.

The *XListInstalledColormaps* function returns a list of the currently installed colormaps for the
screen of the specified window. The order of the colormaps in the list is not significant and is no
explicit indication of the required list. When the allocated list is no longer needed, free it by
using *XFree*.

XListInstalledColormaps can generate a *BadWindow* error.

10.4   Setting and Retrieving the Font Search Path

The set of fonts available from a server depends on a font search path. Xlib provides functions to set and retrieve the search path for a server.

To set the font search path, use *XSetFontPath*.

```
XSetFontPath(display, directories, ndirs)
      Display *display;
      char **directories;
      int ndirs;
```

display   Specifies the connection to the X server.

directories   Specifies the directory path used to look for a font. Setting the path to the empty list restores the default path defined for the X server.

ndirs   Specifies the number of directories in the path.

The *XSetFontPath* function defines the directory search path for font lookup. There is only one search path per X server, not one per client. The encoding and interpretation of the strings is implementation-dependent, but typically they specify directories or font servers to be searched in the order listed. An X server is permitted to cache font information internally, for example, it might cache an entire font from a file and not check on subsequent opens of that font to see if the underlying font file has changed. However, when the font path is changed the X server is guaranteed to flush all cached information about fonts for which there currently are no explicit resource IDs allocated. The meaning of an error from this request is implementation-dependent.

XSetFontPath can generate a *BadValue* error.

To get the current font search path, use *XGetFontPath*.

```
char **XGetFontPath(display, npaths_return)
      Display *display;
      int *npaths_return;
```

display   Specifies the connection to the X server.

npaths_return   Returns the number of strings in the font path array.

The *XGetFontPath* function allocates and returns an array of strings containing the search path. The contents of these strings are implementation-dependent and are not intended to be interpreted by client applications. When it is no longer needed, the data in the font path should be freed by using *XFreeFontPath*.

To free data returned by *XGetFontPath*, use *XFreeFontPath*.

```
XFreeFontPath(list)
      char **list;
```

list   Specifies the array of strings you want to free.

The *XFreeFontPath* function frees the data allocated by *XGetFontPath*.

10.5   Server Grabbing

Xlib provides functions that you can use to grab and ungrab the server. These functions can be used to control processing of output on other connections by the window system server. While the server is grabbed, no processing of requests or close downs on any other connection will occur. A client closing its connection automatically ungrabs the server. Although grabbing the server is highly discouraged, it is sometimes necessary.

To grab the server, use *XGrabServer*.

```
XGrabServer(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XGrabServer* function disables processing of requests and close downs on all other connections than the one this request arrived on. You should not grab the X server any more than is absolutely necessary.

To ungrab the server, use *XUngrabServer*.

```
XUngrabServer(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XUngrabServer* function restarts processing of requests and close downs on other connections. You should avoid grabbing the X server as much as possible.

10.6   Killing Clients

Xlib provides a function to cause the connection to a client to be closed and its resources to be destroyed. To destroy a client, use *XKillClient*.

```
XKillClient(display, resource)
        Display *display;
        XID resource;
```

display   Specifies the connection to the X server.

resource   Specifies any resource associated with the client that you want to destroy or
                         AllTemporary.

The *XKillClient* function forces a close-down of the client that created the resource if a valid resource is specified. If the client has already terminated in either *RetainPermanent* or *RetainTemporary* mode, all of the client's resources are destroyed. If *AllTemporary* is specified, the resources of all clients that have terminated in *RetainTemporary* are destroyed (see Section 3.5). This permits implementation of window manager facilities that aid debugging. A client can set its close-down mode to *RetainTemporary*. If the client then crashes, its windows would not be destroyed. The programmer can then inspect the application's window tree and use the window manager to destroy the zombie windows.

XKillClient can generate a *BadValue* error.

10.7   Screen Saver Control

Xlib provides functions that you can use to set or reset the mode of the screen saver, to force or activate the screen saver, or to obtain the current screen saver values.

To set the screen saver mode, use *XSetScreenSaver*.

```
XSetScreenSaver(display, timeout, interval,
        prefer_blanking, allow_exposures)
    Display *display;
    int timeout, interval;
    int prefer_blanking;
    int allow_exposures;
```

display   Specifies the connection to the X server.

timeout   Specifies the timeout, in seconds, until the screen saver turns on.

interval   Specifies the interval, in seconds, between screen saver alterations.

prefer_blanking Specifies how to enable screen blanking. You can pass *DontPreferBlanking*, *PreferBlanking* or *DefaultBlanking*.

allow_exposures Specifies the screen save control values. You can pass *DontAllowExposures*, *AllowExposures* or *DefaultExposures*.

Timeout and interval are specified in seconds. A timeout of 0 disables the screen saver (but an activated screen saver is not deactivated), and a timeout of −1 restores the default. Other negative values generate a *BadValue* error. If the timeout value is nonzero, *XSetScreenSaver* enables the screen saver. An interval of 0 disables the random-pattern motion. If no input from devices (keyboard, mouse, and so on) is generated for the specified number of timeout seconds once the screen saver is enabled, the screen saver is activated.

For each screen, if blanking is preferred and the hardware supports video blanking, the screen simply goes blank. Otherwise, if either exposures are allowed or the screen can be regenerated without sending *Expose* events to clients, the screen is tiled with the root window background tile randomly re-origined each interval minutes. Otherwise, the screens' state do not change, and the screen saver is not activated. The screen saver is deactivated, and all screen states are restored at the next keyboard or pointer input or at the next call to *XForceScreenSaver* with mode *ScreenSaverReset*.

If the server-dependent screen saver method supports periodic change, the interval argument serves as a hint about how long the change period should be, and zero hints that no periodic change should be made. Examples of ways to change the screen include scrambling the colormap periodically, moving an icon image around the screen periodically, or tiling the screen with the root window background tile, randomly re-origined periodically.

XSetScreenSaver can generate a *BadValue* error.

To force the screen saver on or off, use *XForceScreenSaver*.

```
XForceScreenSaver(display, mode)
    Display *display;
    int mode;
```

display   Specifies the connection to the X server.

mode   Specifies the mode that is to be applied. You can pass *ScreenSaverActive* or *ScreenSaverReset*.

If the specified mode is *ScreenSaverActive* and the screen saver currently is deactivated, *XForceScreenSaver* activates the screen saver even if the screen saver had been disabled with a timeout of zero. If the specified mode is *ScreenSaverReset* and the screen saver currently is enabled, *XForceScreenSaver* deactivates the screen saver if it was activated, and the activation timer is reset to its initial state (as if device input had been received).

XForceScreenSaver can generate a *BadValue* error.

To activate the screen saver, use *XActivateScreenSaver*.

```
XActivateScreenSaver(display)
      Display *display;
```

display   Specifies the connection to the X server.

To reset the screen saver, use *XResetScreenSaver*.

```
XResetScreenSaver(display)
      Display *display;
```

display   Specifies the connection to the X server.

To get the current screen saver values, use *XGetScreenSaver*.

```
XGetScreenSaver(display, timeout_return, interval_return,
          prefer_blanking_return, allow_exposures_return)
      Display *display;
      int *timeout_return, *interval_return;
      int *prefer_blanking_return;
      int *allow_exposures_return;
```

display   Specifies the connection to the X server.

timeout_return   Returns the timeout, in seconds, until the screen saver turns on.

interval_return Returns the interval between screen saver invocations.

prefer_blanking_return

Returns the current screen blanking preference (*DontPreferBlanking*, *PreferBlanking* or *DefaultBlanking*).

allow_exposures_return

Returns the current screen save control value (*DontAllowExposures*, *AllowExposures* or *DefaultExposures*).

10.8   Controlling Host Access

This section discusses how to:

- add, get or remove hosts from the access control list

- change, enable or disable access.

X does not provide any protection on a per-window basis. If you find out the resource ID of a resource, you can manipulate it. To provide some minimal level of protection, however, connections are permitted only from machines you trust. This is adequate on single-user workstations but obviously breaks down on timesharing machines. Although provisions exist in the X protocol for proper connection authentication, the lack of a standard authentication server leaves host-level access control as the only common mechanism.

The initial set of hosts allowed to open connections typically consists of:

- the host the window system is running on

- on POSIX-conformant systems, each host listed in the */etc/X?.hosts* file. The ? indicates the number of the display. This file should consist of host names separated by newlines. DECnet nodes must terminate in :: to distinguish them from Internet hosts.

If a host is not in the access control list when the access control mechanism is enabled and if the host attempts to establish a connection, the server refuses the connection. To change the access list, the client must reside on the same host as the server and/or must have been granted permission in the initial authorization at connection setup.

Servers also can implement other access control policies in addition to or in place of this host access facility. For further information about other access control implementations, see the **X Window System Protocol** specification.

10.8.1   Adding, Getting or Removing Hosts

Xlib provides functions that you can use to add, get, or remove hosts from the access control list. All the host access control functions use the *XHostAddress* structure, which contains:

```
typedef struct {
      int family;     /* for example FamilyInternet */
      int length;     /* length of address, in bytes */
      char *address;  /* pointer to where to find the address */
} XHostAddress;
```

The family member specifies which protocol address family to use (for example, TCP/IP or DECnet) and can be *FamilyInternet*, *FamilyDECnet* or *FamilyChaos*. The length member specifies the length of the address in bytes. The address member specifies a pointer to the address.

For TCP/IP, the address should be in network byte order. For the DECnet family, the server performs no automatic swapping on the address bytes. A Phase IV address is two bytes long. The first byte contains the least-significant eight bits of the node number. The second byte contains the most-significant two bits of the node number in the least-significant two bits of the byte and the area in the most-significant six bits of the byte.

To add a single host, use *XAddHost*.

```
XAddHost(display, host)
      Display *display;
      XHostAddress *host;
```

display   Specifies the connection to the X server.

host   Specifies the host that is to be added.

The *XAddHost* function adds the specified host to the access control list for that display. The server must be on the same host as the client issuing the command, or a *BadAccess* error results.

XAddHost can generate *BadAccess* and *BadValue* errors.

To add multiple hosts at one time, use *XAddHosts*.

```
XAddHosts(display, hosts, num_hosts)
     Display *display;
     XHostAddress *hosts;
     int num_hosts;
```

display   Specifies the connection to the X server.

hosts   Specifies each host that is to be added.

num_hosts   Specifies the number of hosts.

The *XAddHosts* function adds each specified host to the access control list for that display. The server must be on the same host as the client issuing the command, or a *BadAccess* error results.

XAddHosts can generate *BadAccess* and *BadValue* errors.

To obtain a host list, use *XListHosts*.

```
XHostAddress *XListHosts(display, nhosts_return, state_return)
     Display *display;
     int *nhosts_return;
     Bool *state_return;
```

display   Specifies the connection to the X server.

nhosts_return   Returns the number of hosts currently in the access control list.

state_return   Returns the state of the access control.

The *XListHosts* function returns the current access control list as well as whether the use of the list at connection setup was enabled or disabled. *XListHosts* allows a program to find out what machines can make connections. It also returns a pointer to a list of host structures that were allocated by the function. When no longer needed, this memory should be freed by calling *XFree*.

To remove a single host, use *XRemoveHost*.

```
XRemoveHost(display, host)
     Display *display;
     XHostAddress *host;
```

display   Specifies the connection to the X server.

host   Specifies the host that is to be removed.

The *XRemoveHost* function removes the specified host from the access control list for that display. The server must be on the same host as the client process, or a *BadAccess* error results. If you remove your machine from the access list, you can no longer connect to that server, and this operation cannot be reversed unless you reset the server.

XRemoveHost can generate *BadAccess* and *BadValue* errors.

To remove multiple hosts at one time, use *XRemoveHosts*.

```
XRemoveHosts(display, hosts, num_hosts)
      Display *display;
      XHostAddress *hosts;
      int num_hosts;
```

display   Specifies the connection to the X server.

hosts   Specifies each host that is to be removed.

num_hosts   Specifies the number of hosts.

The *XRemoveHosts* function removes each specified host from the access control list for that display. The X server must be on the same host as the client process, or a *BadAccess* error results. If you remove your machine from the access list, you can no longer connect to that server, and this operation cannot be reversed unless you reset the server.

XRemoveHosts can generate *BadAccess* and *BadValue* errors.

10.8.2   Changing, Enabling or Disabling Access Control

Xlib provides functions that you can use to enable, disable or change access control.

For these functions to execute successfully, the client application must reside on the same host as the X server and/or have been given permission in the initial authorization at connection setup.

To change access control, use *XSetAccessControl*.

```
XSetAccessControl(display, mode)
      Display *display;
      int mode;
```

display   Specifies the connection to the X server.

mode   Specifies the mode. You can pass *EnableAccess* or *DisableAccess*.

The *XSetAccessControl* function either enables or disables the use of the access control list at each connection setup.

XSetAccessControl can generate *BadAccess* and *BadValue* errors.

To enable access control, use *XEnableAccessControl*.

```
XEnableAccessControl(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XEnableAccessControl* function enables the use of the access control list at each connection setup.

XEnableAccessControl can generate a *BadAccess* error.

To disable access control, use *XDisableAccessControl*.

```
XDisableAccessControl(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XDisableAccessControl* function disables the use of the access control list at each connection setup.

XDisableAccessControl can generate a *BadAccess* error.

Events

A client application communicates with the X server through the connection you establish with the *XOpenDisplay* function. A client application sends requests to the X server over this connection. These requests are made by the Xlib functions that are called in the client application. Many Xlib functions cause the X server to generate events, and the user's typing or moving the pointer can generate events asynchronously. The X server returns events to the client on the same connection.

This chapter discusses the following topics associated with events:

- event types
- event structures
- event mask
- event processing.

Functions for handling events are dealt with in the next chapter.

11.1 Event Types

An event is data generated asynchronously by the X server as a result of some device activity or as side effects of a request sent by an Xlib function. Device-related events propagate from the source window to ancestor windows until some client application has selected that event type or until the event is explicitly discarded. The X server generally sends an event to a client application only if the client has specifically asked to be informed of that event type, typically by setting the event-mask attribute of the window. The mask can also be set when you create a window or by changing the window's event-mask. You can also mask out events that would propagate to ancestor windows by manipulating the do-not-propagate mask of the window's attributes. However, *MappingNotify* events are always sent to all clients.

An event type describes a specific event generated by the X server. For each event type, a corresponding constant name is defined in <**X11/X.h**>, which is used when referring to an event type. The following table lists the event category and its associated event type or types. The processing associated with these events is discussed in Section 11.5.

Event Category	Event Type
Keyboard events	*KeyPress, KeyRelease*
Pointer events	*ButtonPress, ButtonRelease, MotionNotify*
Window crossing events	*EnterNotify, LeaveNotify*
Input focus events	*FocusIn, FocusOut*
Keymap state notification event	*KeymapNotify*
Exposure events	*Expose, GraphicsExpose, NoExpose*
Structure control events	*CirculateRequest, ConfigureRequest, MapRequest, ResizeRequest*
Window state notification events	*CirculateNotify, ConfigureNotify, CreateNotify, DestroyNotify, GravityNotify, MapNotify, MappingNotify, ReparentNotify, UnmapNotify, VisibilityNotify*
Colormap state notification event	*ColormapNotify*
Client communication events	*ClientMessage, PropertyNotify, SelectionClear, SelectionNotify, SelectionRequest*

11.2   Event Structures

For each event type, a corresponding structure is declared in **<X11/Xlib.h>**. All the event structures have the following common members:

```
typedef struct {
      int type;
      unsigned long serial; /* # of last request processed by
                               server */
      Bool send_event;       /* true if this came from a SendEvent
                               request */
      Display *display;      /* Display the event was read from */
      Window window;
} XAnyEvent;
```

The type member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a *GraphicsExpose* event to a client application, it sends an *XGraphicsExposeEvent* structure with the type member set to *GraphicsExpose*. The display member is set to a pointer to the display the event was read on. The send_event member is set to *True* if the event came from a *SendEvent* protocol request. The serial member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The window member is set to the window that is most useful to toolkit dispatchers.

The X server can send events at any time in the input stream. Xlib stores any events received while waiting for a reply in an event queue for later use. Xlib also provides functions that allow you to check events in the event queue (see Section 12.3).

In addition to the individual structures declared for each event type, the *XEvent* structure is a union of the individual structures declared for each event type. Depending on the type, you should access members of each event by using the *XEvent* union.

```
typedef union _XEvent {
        int type;       /* must not be changed */
        XAnyEvent xany;
        XKeyEvent xkey;
        XButtonEvent xbutton;
        XMotionEvent xmotion;
        XCrossingEvent xcrossing;
        XFocusChangeEvent xfocus;
        XExposeEvent xexpose;
        XGraphicsExposeEvent xgraphicsexpose;
        XNoExposeEvent xnoexpose;
        XVisibilityEvent xvisibility;
        XCreateWindowEvent xcreatewindow;
        XDestroyWindowEvent xdestroywindow;
        XUnmapEvent xunmap;
        XMapEvent xmap;
        XMapRequestEvent xmaprequest;
        XReparentEvent xreparent;
        XConfigureEvent xconfigure;
        XGravityEvent xgravity;
        XResizeRequestEvent xresizerequest;
        XConfigureRequestEvent xconfigurerequest;
        XCirculateEvent xcirculate;
        XCirculateRequestEvent xcirculaterequest;
        XPropertyEvent xproperty;
        XSelectionClearEvent xselectionclear;
        XSelectionRequestEvent xselectionrequest;
        XSelectionEvent xselection;
        XColormapEvent xcolormap;
        XClientMessageEvent xclient;
        XMappingEvent xmapping;
        XErrorEvent xerror;
        XKeymapEvent xkeymap;
        long pad[24];
} XEvent;
```

An *XEvent* structure's first entry always is the type member, which is set to the event type. The second member always is the serial number of the protocol request that generated the event. The third member always is send_event, which is a *Bool* that indicates if the event was sent by a different client. The fourth member always is a display, which is the display that the event was read from. Except for keymap events, the fifth member always is a window, which has been carefully selected to be useful to toolkit dispatchers. To avoid breaking toolkits, the order of these first five entries is not to change. Most events also contain a time member, which is the time at which an event occurred. In addition, a pointer to the generic event must be cast before it is used to access any other information in the structure.

11.3   Event Masks

Clients select event reporting of most events relative to a window. To do this, pass an event mask to an Xlib event-handling function that takes an event_mask argument. The bits of the event mask are defined in <**X11/X.h**>. Each bit in the event mask maps to an event mask name, which describes the event or events you want the X server to return to a client application.

Unless the client has specifically asked for them, most events are not reported to clients when they are generated. Unless the client suppresses them by setting graphics-exposures in the GC to *False*, *GraphicsExpose* and *NoExpose* are reported by default as a result of *XCopyPlane* and *XCopyArea*. *SelectionClear*, *SelectionRequest*, *SelectionNotify* or *ClientMessage* cannot be masked. Selection related events are only sent to clients cooperating with selections (see Section 5.5). When the keyboard or pointer mapping is changed, *MappingNotify* is always sent to clients.

The following table lists the event mask constants you can pass to the event_mask argument and the circumstances in which you would want to specify the event mask:

Event Mask	Circumstances
NoEventMask	No events wanted
KeyPressMask	Keyboard down events wanted
KeyReleaseMask	Keyboard up events wanted
ButtonPressMask	Pointer button down events wanted
ButtonReleaseMask	Pointer button up events wanted
EnterWindowMask	Pointer window entry events wanted
LeaveWindowMask	Pointer window leave events wanted
PointerMotionMask	Pointer motion events wanted
PointerMotionHintMask	Pointer motion hints wanted
Button1MotionMask	Pointer motion while button 1 down
Button2MotionMask	Pointer motion while button 2 down
Button3MotionMask	Pointer motion while button 3 down
Button4MotionMask	Pointer motion while button 4 down
Button5MotionMask	Pointer motion while button 5 down
ButtonMotionMask	Pointer motion while any button down
KeymapStateMask	Keyboard state wanted at window entry and focus in
ExposureMask	Any exposure wanted
VisibilityChangeMask	Any change in visibility wanted
StructureNotifyMask	Any change in window structure wanted
ResizeRedirectMask	Redirect resize of this window
SubstructureNotifyMask	Substructure notification wanted
SubstructureRedirectMask	Redirect structure requests on children
FocusChangeMask	Any change in input focus wanted
PropertyChangeMask	Any change in property wanted
ColormapChangeMask	Any change in colormap wanted
OwnerGrabButtonMask	Automatic grabs should activate with owner_events set to True

11.4   Event Processing Overview

The event reported to a client application during event processing depends on which event masks you provide as the event-mask attribute for a window. For some event masks, there is a one-to-one correspondence between the event mask constant and the event type constant. For example, if you pass the event mask *ButtonPressMask*, the X server sends back only *ButtonPress* events. Most events contain a time member, which is the time at which an event occurred.

In other cases, one event mask constant can map to several event type constants. For example, if you pass the event mask *SubstructureNotifyMask*, the X server can send back *CirculateNotify*, *ConfigureNotify*, *CreateNotify*, *DestroyNotify*, *GravityNotify*, *MapNotify*, *ReparentNotify* or *UnmapNotify* events.

In another case, two event masks can map to one event type. For example, if you pass either *PointerMotionMask* or *ButtonMotionMask*, the X server sends back a *MotionNotify* event.

The following table lists the event mask, its associated event type or types, and the structure name associated with the event type. Some of these structures actually are typedefs to a generic structure that is shared between two event types. Note that N/A appears in columns for which the information is not applicable.

Event Mask	Event Type	Structure	Generic Structure
ButtonMotionMask Button1MotionMask Button2MotionMask Button3MotionMask Button4MotionMask Button5MotionMask	MotionNotify	XPointerMovedEvent	XMotionEvent
ButtonPressMask	ButtonPress	XButtonPressedEvent	XButtonEvent
ButtonReleaseMask	ButtonRelease	XButtonReleasedEvent	XButtonEvent
ColormapChangeMask	ColormapNotify	XColormapEvent	
EnterWindowMask	EnterNotify	XEnterWindowEvent	XCrossingEvent
LeaveWindowMask	LeaveNotify	XLeaveWindowEvent	XCrossingEvent
ExposureMask GCGraphicsExposures in GC	Expose GraphicsExpose NoExpose	XExposeEvent XGraphicsExposeEvent XNoExposeEvent	
FocusChangeMask	FocusIn FocusOut	XFocusInEvent XFocusOutEvent	XFocusChangeEvent XFocusChangeEvent
KeymapStateMask	KeymapNotify	XKeymapEvent	
KeyPressMask KeyReleaseMask	KeyPress KeyRelease	XKeyPressedEvent XKeyReleasedEvent	XKeyEvent XKeyEvent
OwnerGrabButtonMask	N/A	N/A	
PointerMotionMask	MotionNotify	XPointerMovedEvent	XMotionEvent

Event Mask	Event Type	Structure	Generic Structure
PointerMotionHintMask	N/A	N/A	
PropertyChangeMask	PropertyNotify	XPropertyEvent	
ResizeRedirectMask	ResizeRequest	XResizeRequestEvent	
StructureNotifyMask	CirculateNotify ConfigureNotify DestroyNotify GravityNotify MapNotify ReparentNotify UnmapNotify	XCirculateEvent XConfigureEvent XDestroyWindowEvent XGravityEvent XMapEvent XReparentEvent XUnmapEvent	
SubstructureNotifyMask	CirculateNotify ConfigureNotify CreateNotify DestroyNotify GravityNotify MapNotify ReparentNotify UnmapNotify	XCirculateEvent XConfigureEvent XCreateWindowEvent XDestroyWindowEvent XGravityEvent XMapEvent XReparentEvent XUnmapEvent	
SubstructureRedirectMask	CirculateRequest ConfigureRequest MapRequest	XCirculateRequestEvent XConfigureRequestEvent XMapRequestEvent	
N/A	ClientMessage	XClientMessageEvent	
N/A	MappingNotify	XMappingEvent	
N/A	SelectionClear	XSelectionClearEvent	
N/A	SelectionNotify	XSelectionEvent	
N/A	SelectionRequest	XSelectionRequestEvent	
VisibilityChangeMask	VisibilityNotify	XVisibilityEvent	

The sections that follow describe the processing that occurs when you select the different event masks. The sections are organized according to these processing categories:

- keyboard and pointer events
- window crossing events
- input focus events
- keymap state notification events
- exposure events
- window state notification events
- structure control events
- colormap state notification events
- client communication events.

11.5   Keyboard and Pointer Events

This section discusses:

- pointer button events

- keyboard and pointer events.

11.5.1   Pointer Button Events

The following describes the event processing that occurs when a pointer button press is processed with the pointer in some window w and when no active pointer grab is in progress.

The X server searches the ancestors of w from the root down, looking for a passive grab to activate. If no matching passive grab on the button exists, the X server automatically starts an active grab for the client receiving the event and sets the last-pointer-grab time to the current server time. The effect is essentially equivalent to an *XGrabButton* with these client passed arguments:

Argument	Value
w	The event window
event_mask	The client's selected pointer events on the event window
pointer_mode	*GrabModeAsync*
keyboard_mode	*GrabModeAsync*
owner_events	*True*, if the client has selected *OwnerGrabButtonMask* on the event window, otherwise *False*
confine_to	*None*
cursor	*None*

The active grab is automatically terminated when the logical state of the pointer has all buttons released. Clients can modify the active grab by calling *XUngrabPointer* and *XChangeActivePointerGrab*.

11.5.2   Keyboard and Pointer Events

This section discusses the processing that occurs for the keyboard events *KeyPress* and *KeyRelease* and the pointer events *ButtonPress*, *ButtonRelease* and *MotionNotify*. For information about the keyboard event-handling utilities, see Chapter 12.

The X server reports *KeyPress* or *KeyRelease* events to clients wanting information about keys that logically change state. Note that these events are generated for all keys, even those mapped to modifier bits. The X server reports *ButtonPress* or *ButtonRelease* events to clients wanting information about buttons that logically change state.

The X server reports *MotionNotify* events to clients wanting information about when the pointer logically moves. The X server generates this event whenever the pointer is moved and the pointer motion begins and ends in the window. The granularity of *MotionNotify* events is not guaranteed, but a client that selects this event type is guaranteed to receive at least one event when the pointer moves and then rests.

The generation of the logical changes lags the physical changes if device event processing is frozen.

To receive *KeyPress*, *KeyRelease*, *ButtonPress* and *ButtonRelease* events, set *KeyPressMask*, *KeyReleaseMask*, *ButtonPressMask* and *ButtonReleaseMask* bits in the event-mask attribute of the

window.

To receive *MotionNotify* events, set one or more of the following event masks bits in the event-mask attribute of the window.

- *Button1MotionMask* − *Button5MotionMask*

  The client application receives *MotionNotify* events only when one or more of the specified buttons is pressed.

- *ButtonMotionMask*

  The client application receives *MotionNotify* events only when at least one button is pressed.

- *PointerMotionMask*

  The client application receives *MotionNotify* events independent of the state of the pointer buttons.

- *PointerMotionHintMask*

  If *PointerMotionHintMask* is selected in combination with one or more of the above masks, the X server is free to send only one *MotionNotify* event (with the is_hint member of the *XPointerMovedEvent* structure set to *NotifyHint*) to the client for the event window, until either the key or button state changes, the pointer leaves the event window, or the client calls *XQueryPointer* or *XGetMotionEvents*. The server still may send *MotionNotify* events without is_hint set to *NotifyHint*.

The source of the event is the viewable window that the pointer is in. The window used by the X server to report these events depends on the window's position in the window hierarchy and whether any intervening window prohibits the generation of these events. Starting with the source window, the X server searches up the window hierarchy until it locates the first window specified by a client as having an interest in these events. If one of the intervening windows has its do-not-propagate-mask set to prohibit generation of the event type, the events of those types will be suppressed. Clients can modify the actual window used for reporting by performing active grabs and, in the case of keyboard events, by using the focus window.

The structures for these event types contain:

```
typedef struct {
      int type;             /* ButtonPress or ButtonRelease */
      unsigned long serial; /* # of last request processed by
                               server */
      Bool send_event;      /* true if this came from a SendEvent
                               request */
      Display *display;     /* Display the event was read from */
      Window window;        /* ``event'' window it is reported
                               relative to */
      Window root;          /* root window that the event occurred
                               on */
      Window subwindow;     /* child window */
      Time time;            /* milliseconds */
      int x, y;             /* pointer x, y coordinates in event
                               window */
      int x_root, y_root;   /* coordinates relative to root */
      unsigned int state;   /* key or button mask */
      unsigned int button;  /* detail */
      Bool same_screen;     /* same screen flag */
} XButtonEvent;
typedef XButtonEvent XButtonPressedEvent;
typedef XButtonEvent XButtonReleasedEvent;

typedef struct {
      int type;             /* KeyPress or KeyRelease */
      unsigned long serial; /* # of last request processed by
                               server */
      Bool send_event;      /* true if this came from a SendEvent
                               request */
      Display *display;     /* Display the event was read from */
      Window window;        /* ``event'' window it is reported
                               relative to */
      Window root;          /* root window that the event occurred
                               on */
      Window subwindow;     /* child window */
      Time time;            /* milliseconds */
      int x, y;             /* pointer x, y coordinates in event
                               window */
      int x_root, y_root;   /* coordinates relative to root */
      unsigned int state;   /* key or button mask */
      unsigned int keycode; /* detail */
      Bool same_screen;     /* same screen flag */
} XKeyEvent;
typedef XKeyEvent XKeyPressedEvent;
typedef XKeyEvent XKeyReleasedEvent;
```

```
typedef struct {
      int type;              /* MotionNotify */
      unsigned long serial;  /* # of last request processed by
                                server */
      Bool send_event;       /* true if this came from a SendEvent
                                request */
      Display *display;      /* Display the event was read from */
      Window window;         /* ``event'' window reported
                                relative to */
      Window root;           /* root window that the event occurred
                                on */
      Window subwindow;      /* child window */
      Time time;             /* milliseconds */
      int x, y;              /* pointer x, y coordinates in event
                                window */
      int x_root, y_root;    /* coordinates relative to root */
      unsigned int state;    /* key or button mask */
      char is_hint;          /* detail */
      Bool same_screen;      /* same screen flag */
} XMotionEvent;
typedef XMotionEvent XPointerMovedEvent;
```

These structures have the following common members: window, root, subwindow, time, x, y, x_root, y_root, state and same_screen. The window member is set to the window on which the event was generated and is referred to as the event window. As long as the conditions previously discussed are met, this is the window used by the X server to report the event. The root member is set to the source window's root window. The x_root and y_root members are set to the pointer's coordinates relative to the root window's origin at the time of the event.

The same_screen member is set to indicate whether the event window is on the same screen as the root window and can be either *True* or *False*. If *True*, the event and root windows are on the same screen. If *False*, the event and root windows are not on the same screen.

If the source window is an inferior of the event window, the subwindow member of the structure is set to the child of the event window that is the source window or the child of the event window that is an ancestor of the source window. Otherwise, the X server sets the subwindow member to *None*. The time member is set to the time when the event was generated and is expressed in milliseconds.

If the event window is on the same screen as the root window, the x and y members are set to the coordinates relative to the event window's origin. Otherwise, these members are set to zero.

The state member is set to indicate the logical state of the pointer buttons and modifier keys just prior to the event, which is the bitwise inclusive OR of one or more of the button or modifier key masks: *Button1Mask*, *Button2Mask*, *Button3Mask*, *Button4Mask*, *Button5Mask*, *ShiftMask*, *LockMask*, *ControlMask*, *Mod1Mask*, *Mod2Mask*, *Mod3Mask*, *Mod4Mask* and *Mod5Mask*.

Each of these structures also has a member that indicates the detail. For the *XKeyPressedEvent* and *XKeyReleasedEvent* structures, this member is called keycode. It is set to a number that represents a physical key on the keyboard. The keycode is an arbitrary representation for any key on the keyboard (see Section 13.7 and Section 17.1).

For the *XButtonPressedEvent* and *XButtonReleasedEvent* structures, this member is called button. It represents the pointer button that changed state and can be the *Button1*, *Button2*, *Button3*, *Button4* or *Button5* value. For the *XPointerMovedEvent* structure, this member is called is_hint. It can be set to *NotifyNormal* or *NotifyHint*.

11.6 Window Entry/Exit Events

This section describes the processing that occurs for the window crossing events *EnterNotify* and *LeaveNotify*. If a pointer motion or a window hierarchy change causes the pointer to be in a different window than before, the X server reports *EnterNotify* or *LeaveNotify* events to clients who have selected for these events. All *EnterNotify* and *LeaveNotify* events caused by a hierarchy change are generated after any hierarchy event (*UnmapNotify*, *MapNotify*, *ConfigureNotify*, *GravityNotify*, *CirculateNotify*) caused by that change; however, the X protocol does not constrain the ordering of *EnterNotify* and *LeaveNotify* events with respect to *FocusOut*, *VisibilityNotify* and *Expose* events.

This contrasts with *MotionNotify* events, which are also generated when the pointer moves but only when the pointer motion begins and ends in a single window. An *EnterNotify* or *LeaveNotify* event also can be generated when some client application calls *XGrabPointer* and *XUngrabPointer*.

To receive *EnterNotify* or *LeaveNotify* events, set the *EnterWindowMask* or *LeaveWindowMask* bits of the event-mask attribute of the window.

The structure for these event types contains:

```
typedef struct {
        int type;               /* EnterNotify or LeaveNotify */
        unsigned long serial; /* # of last request processed by
                                  server */
        Bool send_event;        /* true if this came from a SendEvent
                                  request */
        Display *display;       /* Display the event was read from */
        Window window;          /* ''event'' window reported
                                  relative to */
        Window root;            /* root window that the event occurred
                                  on */
        Window subwindow;       /* child window */
        Time time;              /* milliseconds */
        int x, y;               /* pointer x, y coordinates in event
                                  window */
        int x_root, y_root;     /* coordinates relative to root */
        int mode;               /* NotifyNormal, NotifyGrab, NotifyUngrab */
        int detail;
                                /*
                                 * NotifyAncestor, NotifyVirtual,
                                 * NotifyInferior, NotifyNonlinear,
                                 * NotifyNonlinearVirtual
                                 */
        Bool same_screen;       /* same screen flag */
        Bool focus;             /* boolean focus */
        unsigned int state;     /* key or button mask */
} XCrossingEvent;
typedef XCrossingEvent XEnterWindowEvent;
typedef XCrossingEvent XLeaveWindowEvent;
```

The window member is set to the window on which the *EnterNotify* or *LeaveNotify* event was generated and is referred to as the event window. This is the window used by the X server to report the event, and is relative to the root window on which the event occurred. The root member is set to the root window of the screen on which the event occurred.

For a *LeaveNotify* event, if a child of the event window contains the initial position of the pointer, the subwindow component is set to that child. Otherwise, the X server sets the subwindow member to *None*. For an *EnterNotify* event, if a child of the event window contains the final pointer position, the subwindow component is set to that child or *None*.

The time member is set to the time when the event was generated and is expressed in milliseconds. The x and y members are set to the coordinates of the pointer position in the event window. This position is always the pointer's final position, not its initial position. If the event window is on the same screen as the root window, x and y are the pointer coordinates relative to the event window's origin. Otherwise, x and y are set to zero. The x_root and y_root members are set to the pointer's coordinates relative to the root window's origin at the time of the event.

The same_screen member is set to indicate whether the event window is on the same screen as the root window and can be either *True* or *False*. If *True*, the event and root windows are on the same screen. If *False*, the event and root windows are not on the same screen.

The focus member is set to indicate whether the event window is the focus window or an inferior of the focus window. The X server can set this member to either *True* or *False*. If *True*, the event window is the focus window or an inferior of the focus window. If *False*, the event window is not the focus window or an inferior of the focus window.

The state member is set to indicate the state of the pointer buttons and modifier keys just prior to the event. The X server can set this member to the bitwise inclusive OR of one or more of the button or modifier key masks: *Button1Mask*, *Button2Mask*, *Button3Mask*, *Button4Mask*, *Button5Mask*, *ShiftMask*, *LockMask*, *ControlMask*, *Mod1Mask*, *Mod2Mask*, *Mod3Mask*, *Mod4Mask*, *Mod5Mask*.

The mode member is set to indicate whether the events are normal events, pseudo-motion events when a grab activates, or pseudo-motion events when a grab deactivates. The X server can set this member to *NotifyNormal*, *NotifyGrab* or *NotifyUngrab*.

The detail member is set to indicate the notify detail and can be *NotifyAncestor*, *NotifyVirtual*, *NotifyInferior*, *NotifyNonlinear* or *NotifyNonlinearVirtual*.

11.6.1   Normal Entry/Exit Events

EnterNotify and *LeaveNotify* events are generated when the pointer moves from one window to another window. Normal events are identified by *XEnterWindowEvent* or *XLeaveWindowEvent* structures whose mode member is set to *NotifyNormal*.

- When the pointer moves from window A to window B and A is an inferior of B, the X server does the following:

  — It generates a *LeaveNotify* event on window A, with the detail member of the *XLeaveWindowEvent* structure set to *NotifyAncestor*.

  — It generates a *LeaveNotify* event on each window between window A and window B, exclusive, with the detail member of each *XLeaveWindowEvent* structure set to *NotifyVirtual*.

  — It generates an *EnterNotify* event on window B, with the detail member of the *XEnterWindowEvent* structure set to *NotifyInferior*.

- When the pointer moves from window A to window B and B is an inferior of A, the X server does the following:

  — It generates a *LeaveNotify* event on window A, with the detail member of the *XLeaveWindowEvent* structure set to *NotifyInferior*.

— It generates an *EnterNotify* event on each window between window A and window B, exclusive, with the detail member of each *XEnterWindowEvent* structure set to *NotifyVirtual*.

— It generates an *EnterNotify* event on window B, with the detail member of the *XEnterWindowEvent* structure set to *NotifyAncestor*.

- When the pointer moves from window A to window B and window C is their least common ancestor, the X server does the following:

— It generates a *LeaveNotify* event on window A, with the detail member of the *XLeaveWindowEvent* structure set to *NotifyNonlinear*.

— It generates a *LeaveNotify* event on each window between window A and window C, exclusive, with the detail member of each *XLeaveWindowEvent* structure set to *NotifyNonlinearVirtual*.

— It generates an *EnterNotify* event on each window between window C and window B, exclusive, with the detail member of each *XEnterWindowEvent* structure set to *NotifyNonlinearVirtual*.

— It generates an *EnterNotify* event on window B, with the detail member of the *XEnterWindowEvent* structure set to *NotifyNonlinear*.

- When the pointer moves from window A to window B on different screens, the X server does the following:

— It generates a *LeaveNotify* event on window A, with the detail member of the *XLeaveWindowEvent* structure set to *NotifyNonlinear*.

— If window A is not a root window, it generates a *LeaveNotify* event on each window above window A up to and including its root, with the detail member of each *XLeaveWindowEvent* structure set to *NotifyNonlinearVirtual*.

— If window B is not a root window, it generates an *EnterNotify* event on each window from window B's root down to but not including window B, with the detail member of each *XEnterWindowEvent* structure set to *NotifyNonlinearVirtual*.

— It generates an *EnterNotify* event on window B, with the detail member of the *XEnterWindowEvent* structure set to *NotifyNonlinear*.

11.6.2 Grab and Ungrab Entry/Exit Events

Pseudo-motion mode *EnterNotify* and *LeaveNotify* events are generated when a pointer grab activates or deactivates. Events in which the pointer grab activates are identified by *XEnterWindowEvent* or *XLeaveWindowEvent* structures whose mode member is set to *NotifyGrab*. Events in which the pointer grab deactivates are identified by *XEnterWindowEvent* or *XLeaveWindowEvent* structures whose mode member is set to *NotifyUngrab* (see *XGrabPointer*).

- When a pointer grab activates after any initial warp into a confine_to window and before generating any actual *ButtonPress* event that activates the grab, G is the grab_window for the grab, and P is the window the pointer is in, the X server does the following:

— It generates *EnterNotify* and *LeaveNotify* events (see Section 11.6.1) with the mode members of the *XEnterWindowEvent* and *XLeaveWindowEvent* structures set to *NotifyGrab*. These events are generated as if the pointer were to suddenly warp from its current position in P to some position in G. However, the pointer does not warp, and the X server uses the pointer position as both the initial and final positions for the events.

- When a pointer grab deactivates after generating any actual *ButtonRelease* event that deactivates the grab, G is the grab_window for the grab, and P is the window the pointer is in, the X server does the following:

  — It generates *EnterNotify* and *LeaveNotify* events (see Section 11.6.1) with the mode members of the *XEnterWindowEvent* and *XLeaveWindowEvent* structures set to *NotifyUngrab*. These events are generated as if the pointer were to suddenly warp from some position in G to its current position in P. However, the pointer does not warp, and the X server uses the current pointer position as both the initial and final positions for the events.

11.7   Input Focus Events

This section describes the processing that occurs for the input focus events *FocusIn* and *FocusOut*. The X server can report *FocusIn* or *FocusOut* events to clients wanting information about when the input focus changes. The keyboard is always attached to some window (typically, the root window or a top-level window), which is called the focus window. The focus window and the position of the pointer determine the window that receives keyboard input. Clients may need to know when the input focus changes to control highlighting of areas on the screen.

To receive *FocusIn* or *FocusOut* events, set the *FocusChangeMask* bit in the event-mask attribute of the window.

The structure for these event types contains:

```
typedef struct {
        int type;               /* FocusIn or FocusOut */
        unsigned long serial; /* # of last request processed by
                                  server */
        Bool send_event;        /* true if this came from a SendEvent
                                  request */
        Display *display;       /* Display the event was read from */
        Window window;          /* window of event */
        int mode;               /* NotifyNormal, NotifyGrab,
                                  NotifyUngrab */
        int detail;
                                /*
                                 * NotifyAncestor, NotifyVirtual,
                                 * NotifyInferior, NotifyNonlinear,
                                 * NotifyNonlinearVirtual, NotifyPointer,
                                 * NotifyPointerRoot, NotifyDetailNone
                                 */
} XFocusChangeEvent;
typedef XFocusChangeEvent XFocusInEvent;
typedef XFocusChangeEvent XFocusOutEvent;
```

The window member is set to the window on which the *FocusIn* or *FocusOut* event was generated. This is the window used by the X server to report the event. The mode member is set to indicate whether the focus events are normal focus events, focus events while grabbed, focus events when a grab activates, or focus events when a grab deactivates. The X server can set the mode member to *NotifyNormal*, *NotifyWhileGrabbed*, *NotifyGrab* or *NotifyUngrab*.

All *FocusOut* events caused by a window unmap are generated after any *UnmapNotify* event; however, the X protocol does not constrain the ordering of *FocusOut* events with respect to generated *EnterNotify*, *LeaveNotify*, *VisibilityNotify* and *Expose* events.

Depending on the event mode, the detail member is set to indicate the notify detail and can be *NotifyAncestor*, *NotifyVirtual*, *NotifyInferior*, *NotifyNonlinear*, *NotifyNonlinearVirtual*, *NotifyPointer*, *NotifyPointerRoot* or *NotifyDetailNone*.

11.7.1   Normal Focus Events and Focus Events While Grabbed

Normal focus events are identified by *XFocusInEvent* or *XFocusOutEvent* structures whose mode member is set to *NotifyNormal*. Focus events while grabbed are identified by *XFocusInEvent* or *XFocusOutEvent* structures whose mode member is set to *NotifyWhileGrabbed*. The X server processes normal focus and focus events while grabbed according to the following:

- When the focus moves from window A to window B, A is an inferior of B, and the pointer is in window P, the X server does the following:

  — It generates a *FocusOut* event on window A, with the detail member of the *XFocusOutEvent* structure set to *NotifyAncestor*.

  — It generates a *FocusOut* event on each window between window A and window B, exclusive, with the detail member of each *XFocusOutEvent* structure set to *NotifyVirtual*.

  — It generates a *FocusIn* event on window B, with the detail member of the *XFocusOutEvent* structure set to *NotifyInferior*.

  — If window P is an inferior of window B but window P is not window A or an inferior or ancestor of window A, it generates a *FocusIn* event on each window below window B, down to and including window P, with the detail member of each *XFocusInEvent* structure set to *NotifyPointer*.

- When the focus moves from window A to window B, B is an inferior of A, and the pointer is in window P, the X server does the following:

  — If window P is an inferior of window A but P is not an inferior of window B or an ancestor of B, it generates a *FocusOut* event on each window from window P up to but not including window A, with the detail member of each *XFocusOutEvent* structure set to *NotifyPointer*.

  — It generates a *FocusOut* event on window A, with the detail member of the *XFocusOutEvent* structure set to *NotifyInferior*.

  — It generates a *FocusIn* event on each window between window A and window B, exclusive, with the detail member of each *XFocusInEvent* structure set to *NotifyVirtual*.

  — It generates a *FocusIn* event on window B, with the detail member of the *XFocusInEvent* structure set to *NotifyAncestor*.

- When the focus moves from window A to window B, window C is their least common ancestor, and the pointer is in window P, the X server does the following:

  — If window P is an inferior of window A, it generates a *FocusOut* event on each window from window P up to but not including window A, with the detail member of the *XFocusOutEvent* structure set to *NotifyPointer*.

  — It generates a *FocusOut* event on window A, with the detail member of the *XFocusOutEvent* structure set to *NotifyNonlinear*.

  — It generates a *FocusOut* event on each window between window A and window C, exclusive, with the detail member of each *XFocusOutEvent* structure set to *NotifyNonlinearVirtual*.

  — It generates a *FocusIn* event on each window between C and B, exclusive, with the detail member of each *XFocusInEvent* structure set to *NotifyNonlinearVirtual*.

  — It generates a *FocusIn* event on window B, with the detail member of the *XFocusInEvent* structure set to *NotifyNonlinear*.

— If window P is an inferior of window B, it generates a *FocusIn* event on each window below window B down to and including window P, with the detail member of the *XFocusInEvent* structure set to *NotifyPointer*.

- When the focus moves from window A to window B on different screens and the pointer is in window P, the X server does the following:

    — If window P is an inferior of window A, it generates a *FocusOut* event on each window from window P up to but not including window A, with the detail member of each *XFocusOutEvent* structure set to *NotifyPointer*.

    — It generates a *FocusOut* event on window A, with the detail member of the *XFocusOutEvent* structure set to *NotifyNonlinear*.

    — If window A is not a root window, it generates a *FocusOut* event on each window above window A up to and including its root, with the detail member of each *XFocusOutEvent* structure set to *NotifyNonlinearVirtual*.

    — If window B is not a root window, it generates a *FocusIn* event on each window from window B's root down to but not including window B, with the detail member of each *XFocusInEvent* structure set to *NotifyNonlinearVirtual*.

    — It generates a *FocusIn* event on window B, with the detail member of each *XFocusInEvent* structure set to *NotifyNonlinear*.

    — If window P is an inferior of window B, it generates a *FocusIn* event on each window below window B down to and including window P, with the detail member of each *XFocusInEvent* structure set to *NotifyPointer*.

- When the focus moves from window A to *PointerRoot* (events sent to the window under the pointer) or *None* (discard), and the pointer is in window P, the X server does the following:

    — If window P is an inferior of window A, it generates a *FocusOut* event on each window from window P up to but not including window A, with the detail member of each *XFocusOutEvent* structure set to *NotifyPointer*.

    — It generates a *FocusOut* event on window A, with the detail member of the *XFocusOutEvent* structure set to *NotifyNonlinear*.

    — If window A is not a root window, it generates a *FocusOut* event on each window above window A up to and including its root, with the detail member of each *XFocusOutEvent* structure set to *NotifyNonlinearVirtual*.

    — It generates a *FocusIn* event on the root window of all screens, with the detail member of each *XFocusInEvent* structure set to *NotifyPointerRoot* (or *NotifyDetailNone*).

    — If the new focus is *PointerRoot*, it generates a *FocusIn* event on each window from window P's root down to and including window P, with the detail member of each *XFocusInEvent* structure set to *NotifyPointer*.

- When the focus moves from *PointerRoot* (events sent to the window under the pointer) or *None* to window A, and the pointer is in window P, the X server does the following:

    — If the old focus is *PointerRoot*, it generates a *FocusOut* event on each window from window P up to and including window P's root, with the detail member of each *XFocusOutEvent* structure set to *NotifyPointer*.

    — It generates a *FocusOut* event on all root windows, with the detail member of each *XFocusOutEvent* structure set to *NotifyPointerRoot* (or *NotifyDetailNone*).

— If window A is not a root window, it generates a *FocusIn* event on each window from window A's root down to but not including window A, with the detail member of each *XFocusInEvent* structure set to *NotifyNonlinearVirtual*.

— It generates a *FocusIn* event on window A, with the detail member of the *XFocusInEvent* structure set to *NotifyNonlinear*.

— If window P is an inferior of window A, it generates a *FocusIn* event on each window below window A down to and including window P, with the detail member of each *XFocusInEvent* structure set to *NotifyPointer*.

• When the focus moves from *PointerRoot* (events sent to the window under the pointer) to *None* (or *vice versa*), and the pointer is in window P, the X server does the following:

— If the old focus is *PointerRoot*, it generates a *FocusOut* event on each window from window P up to and including window P's root, with the detail member of each *XFocusOutEvent* structure set to *NotifyPointer*.

— It generates a *FocusOut* event on all root windows, with the detail member of each *XFocusOutEvent* structure set to either *NotifyPointerRoot* or *NotifyDetailNone*.

— It generates a *FocusIn* event on all root windows, with the detail member of each *XFocusInEvent* structure set to *NotifyDetailNone* or *NotifyPointerRoot*.

— If the new focus is *PointerRoot*, it generates a *FocusIn* event on each window from window P's root down to and including window P, with the detail member of each *XFocusInEvent* structure set to *NotifyPointer*.

11.7.2   Focus Events Generated by Grabs

Focus events in which the keyboard grab activates are identified by *XFocusInEvent* or *XFocusOutEvent* structures whose mode member is set to *NotifyGrab*. Focus events in which the keyboard grab deactivates are identified by *XFocusInEvent* or *XFocusOutEvent* structures whose mode member is set to *NotifyUngrab* (see *XGrabKeyboard*).

• When a keyboard grab activates before generating any actual *KeyPress* event that activates the grab, G is the grab_window, and F is the current focus, the X server does the following:

— It generates *FocusIn* and *FocusOut* events, with the mode members of the *XFocusInEvent* and *XFocusOutEvent* structures set to *NotifyGrab*. These events are generated as if the focus were to change from F to G.

• When a keyboard grab deactivates after generating any actual *KeyRelease* event that deactivates the grab, G is the grab_window, and F is the current focus, the X server does the following:

— It generates *FocusIn* and *FocusOut* events, with the mode members of the *XFocusInEvent* and *XFocusOutEvent* structures set to *NotifyUngrab*. These events are generated as if the focus were to change from G to F.

11.8   Key Map State Notification Events

The X server can report *KeymapNotify* events to clients that want information about changes in their keyboard state.

To receive *KeymapNotify* events, set the *KeymapStateMask* bit in the event-mask attribute of the window. The X server generates this event immediately after every *EnterNotify* and *FocusIn* event.

The structure for this event type contains:

```
/* generated on EnterWindow and FocusIn when KeymapState selected */
typedef struct {
        int type;               /* KeymapNotify */
        unsigned long serial; /* # of last request processed by
                                  server */
        Bool send_event;      /* true if this came from a SendEvent
                                  request */
        Display *display;     /* Display the event was read from */
        Window window;
        char key_vector[32];
} XKeymapEvent;
```

The window member is not used but is present to aid some toolkits. The key_vector member is set to the bit vector of the keyboard. Each bit set to 1 indicates that the corresponding key is currently pressed. The vector is represented as 32 bytes. Byte N (from 0) contains the bits for keys 8N to 8N + 7 with the least-significant bit in the byte representing key 8N.

11.9   Exposure Events

The X protocol does not guarantee to preserve the contents of window regions when the windows are obscured or reconfigured. Some implementations may preserve the contents of windows. Other implementations are free to destroy the contents of windows when exposed. X expects client applications to assume the responsibility for restoring the contents of an exposed window region. (An exposed window region describes a formerly obscured window whose region becomes visible.) Therefore, the X server sends *Expose* events describing the window and the region of the window that has been exposed. A naive client application usually redraws the entire window. A more sophisticated client application redraws only the exposed region.

11.9.1   Expose Events

The X server can report *Expose* events to clients wanting information about when the contents of window regions have been lost. The circumstances in which the X server generates *Expose* events are not as definite as those for other events. However, the X server never generates *Expose* events on windows whose class you specified as *InputOnly*. The X server can generate *Expose* events when no valid contents are available for regions of a window and either the regions are visible, the regions are viewable and the server is (perhaps newly) maintaining backing store on the window, or the window is not viewable but the server is (perhaps newly) honouring the window's backing-store attribute of *Always* or *WhenMapped*. The regions decompose into an (arbitrary) set of rectangles, and an *Expose* event is generated for each rectangle. For any given window, the X server guarantees to report contiguously all of the regions exposed by some action that causes *Expose* events, such as raising a window.

To receive *Expose* events, set the *ExposureMask* bit in the event-mask attribute of the window.

The structure for this event type contains:

```
typedef struct {
        int type;               /* Expose */
        unsigned long serial; /* # of last request processed by
                                    server */
        Bool send_event;       /* true if this came from a SendEvent
                                    request */
        Display *display;      /* Display the event was read from */
        Window window;
        int x, y;
        int width, height;
        int count;             /* if non-zero, at least this many
                                    more */
} XExposeEvent;
```

The window member is set to the exposed (damaged) window. The x and y members are set to the coordinates relative to the window's origin and indicate the upper-left corner of the rectangle. The width and height members are set to the size (extent) of the rectangle. The count member is set to the number of *Expose* events that are to follow. If count is zero, no more *Expose* events follow for this window. However, if count is non-zero, at least that number of *Expose* events (and possibly more) follow for this window. Simple applications that do not want to optimize redisplay by distinguishing between subareas of its window can just ignore all *Expose* events with non-zero counts and perform full redisplays on events with zero counts.

11.9.2 GraphicsExpose and NoExpose Events

The X server can report *GraphicsExpose* events to clients wanting information about when a destination region could not be computed during certain graphics requests: *XCopyArea* or *XCopyPlane*. The X server generates this event whenever a destination region could not be computed due to an obscured or out-of-bounds source region. In addition, the X server guarantees to report contiguously all of the regions exposed by some graphics request (for example, copying an area of a drawable to a destination drawable).

The X server generates a *NoExpose* event whenever a graphics request that might produce a *GraphicsExpose* event does not produce any. In other words, the client is really asking for a *GraphicsExpose* event but instead receives a *NoExpose* event.

To receive *GraphicsExpose* or *NoExpose* events, you must first set the graphics-exposure attribute of the graphics context to *True*. You also can set the graphics-expose attribute when creating a graphics context using *XCreateGC* or by calling *XSetGraphicsExposures*.

The structures for these event types contain:

```
typedef struct {
        int type;               /* GraphicsExpose */
        unsigned long serial;   /* # of last request processed by
                                        server */
        Bool send_event;        /* true if this came from a SendEvent
                                        request */
        Display *display;       /* Display the event was read from */
        Drawable drawable;
        int x, y;
        int width, height;
        int count;              /* if non-zero, at least this many
                                        more */
        int major_code;         /* core is CopyArea or CopyPlane */
        int minor_code;         /* not defined in the core */
} XGraphicsExposeEvent;

typedef struct {
        int type;               /* NoExpose */
        unsigned long serial;   /* # of last request processed by
                                        server */
        Bool send_event;        /* true if this came from a SendEvent
                                        request */
        Display *display;       /* Display the event was read from */
        Drawable drawable;
        int major_code;         /* core is CopyArea or CopyPlane */
        int minor_code;         /* not defined in the core */
} XNoExposeEvent;
```

Both structures have these common members: drawable, major_code and minor_code. The drawable member is set to the drawable of the destination region on which the graphics request was to be performed. The major_code member is set to the graphics request initiated by the client and can be either *X_CopyArea* or *X_CopyPlane*. If it is *X_CopyArea*, a call to *XCopyArea* initiated the request. If it is *X_CopyPlane*, a call to *XCopyPlane* initiated the request. These constants are defined in **<X11/Xproto.h>**. The minor_code member, like the major_code member, indicates which graphics request was initiated by the client. However, the minor_code member is not defined by the core X protocol and will be zero in these cases, although it may be used by an extension.

The *XGraphicsExposeEvent* structure has these additional members: x, y, width, height and count.
The x and y members are set to the coordinates relative to the drawable's origin and indicate the
upper-left corner of the rectangle. The width and height members are set to the size (extent) of
the rectangle. The count member is set to the number of *GraphicsExpose* events to follow. If
count is zero, no more *GraphicsExpose* events follow for this window. However, if count is non-
zero, at least that number of *GraphicsExpose* events (and possibly more) are to follow for this
window.

11.10 Window State Change Events

The following sections discuss:

- *CirculateNotify* events
- *ConfigureNotify* events
- *CreateNotify* events
- *DestroyNotify* events
- *GravityNotify* events
- *MapNotify* events
- *MappingNotify* events
- *ReparentNotify* events
- *UnmapNotify* events
- *VisibilityNotify* events.

11.10.1 CirculateNotify Events

The X server can report *CirculateNotify* events to clients wanting information about when a window changes its position in the stack. The X server generates this event type whenever a window is actually restacked as a result of a client application calling *XCirculateSubwindows*, *XCirculateSubwindowsUp* or *XCirculateSubwindowsDown*.

To receive *CirculateNotify* events, set the *StructureNotifyMask* bit in the event-mask attribute of the window or the *SubstructureNotifyMask* bit in the event-mask attribute of the parent window (in which case, circulating any child generates an event).

The structure for this event type contains:

```
typedef struct {
      int type;              /* CirculateNotify */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;       /* true if this came from a SendEvent
                                request */
      Display *display;      /* Display the event was read from */
      Window event;
      Window window;
      int place;             /* PlaceOnTop, PlaceOnBottom */
} XCirculateEvent;
```

The event member is set either to the restacked window or to its parent, depending on whether *StructureNotify* or *SubstructureNotify* was selected. The window member is set to the window that was restacked. The place member is set to the window's position after the restack occurs and is either *PlaceOnTop* or *PlaceOnBottom*. If it is *PlaceOnTop*, the window is now on top of all siblings. If it is *PlaceOnBottom*, the window is now below all siblings.

11.10.2 ConfigureNotify Events

The X server can report *ConfigureNotify* events to clients wanting information about actual changes to a window's state, such as size, position, border and stacking order. The X server generates this event type whenever one of the following configure window requests made by a client application actually completes:

- A window's size, position, border and/or stacking order is reconfigured by calling *XConfigureWindow*.

- The window's position in the stacking order is changed by calling *XLowerWindow*, *XRaiseWindow* or *XRestackWindows*.

- A window is moved by calling *XMoveWindow*.

- A window's size is changed by calling *XResizeWindow*.

- A window's size and location is changed by calling *XMoveResizeWindow*.

- A window is mapped and its position in the stacking order is changed by calling *XMapRaised*.

- A window's border width is changed by calling *XSetWindowBorderWidth*.

To receive *ConfigureNotify* events, set the *StructureNotifyMask* bit in the event-mask attribute of the window or the *SubstructureNotifyMask* bit in the event-mask attribute of the parent window (in which case, configuring any child generates an event).

The structure for this event type contains:

```
typedef struct {
      int type;              /* ConfigureNotify */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;       /* true if this came from a SendEvent
                                request */
      Display *display;      /* Display the event was read from */
      Window event;
      Window window;
      int x, y;
      int width, height;
      int border_width;
      Window above;
      Bool override_redirect;
} XConfigureEvent;
```

The event member is set either to the reconfigured window or to its parent, depending on whether *StructureNotify* or *SubstructureNotify* was selected. The window member is set to the window whose size, position, border and/or stacking order was changed.

The x and y members are set to the coordinates relative to the parent window's origin and indicate the position of the upper-left outside corner of the window. The width and height members are set to the inside size of the window, not including the border. The border_width member is set to the width of the window's border, in pixels.

The above member is set to the sibling window and is used for stacking operations. If the X server sets this member to *None*, the window whose state was changed is on the bottom of the stack with respect to sibling windows. However, if this member is set to a sibling window, the window whose state was changed is placed on top of this sibling window.

The override_redirect member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the override_redirect member is *True*.

11.10.3 CreateNotify Events

The X server can report *CreateNotify* events to clients wanting information about creation of windows. The X server generates this event whenever a client application creates a window by calling *XCreateWindow* or *XCreateSimpleWindow*.

To receive *CreateNotify* events, set the *SubstructureNotifyMask* bit in the event-mask attribute of the window. Creating any children then generates an event.

The structure for the event type contains:

```
typedef struct {
      int type;               /* CreateNotify */
      unsigned long serial;   /* # of last request processed by
                                  server */
      Bool send_event;        /* true if this came from a SendEvent
                                  request */
      Display *display;       /* Display the event was read from */
      Window parent;          /* parent of the window */
      Window window;          /* window id of window created */
      int x, y;               /* window location */
      int width, height;      /* size of window */
      int border_width;       /* border width */
      Bool override_redirect; /* creation should be overridden */
} XCreateWindowEvent;
```

The parent member is set to the created window's parent. The window member specifies the created window. The x and y members are set to the created window's coordinates relative to the parent window's origin and indicate the position of the upper-left outside corner of the created window. The width and height members are set to the inside size of the created window (not including the border) and are always non-zero. The border_width member is set to the width of the created window's border, in pixels. The override_redirect member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the override_redirect member is *True*.

11.10.4 DestroyNotify Events

The X server can report *DestroyNotify* events to clients wanting information about which windows are destroyed. The X server generates this event whenever a client application destroys a window by calling *XDestroyWindow* or *XDestroySubwindows*.

The ordering of the *DestroyNotify* events is such that for any given window, *DestroyNotify* is generated on all inferiors of the window before being generated on the window itself. The X protocol does not constrain the ordering among siblings and across subhierarchies.

To receive *DestroyNotify* events, set the *StructureNotifyMask* bit in the event-mask attribute of the window or the *SubstructureNotifyMask* bit in the event-mask attribute of the parent window (in which case, destroying any child generates an event).

The structure for this event type contains:

```
typedef struct {
      int type;              /* DestroyNotify */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;      /* true if this came from a SendEvent
                                request */
      Display *display;      /* Display the event was read from */
      Window event;
      Window window;
} XDestroyWindowEvent;
```

The event member is set either to the destroyed window or to its parent, depending on whether *StructureNotify* or *SubstructureNotify* was selected. The window member is set to the window that is destroyed.

11.10.5 GravityNotify Events

The X server can report *GravityNotify* events to clients wanting information about when a window is moved because of a change in the size of its parent. The X server generates this event whenever a client application actually moves a child window as a result of resizing its parent by calling *XConfigureWindow*, *XMoveResizeWindow* or *XResizeWindow*.

To receive *GravityNotify* events, set the *StructureNotifyMask* bit in the event-mask attribute of the window or the *SubstructureNotifyMask* bit in the event-mask attribute of the parent window (in which case, any child that is moved because its parent has been resized generates an event).

The structure for this event type contains:

```
typedef struct {
      int type;              /* GravityNotify */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;      /* true if this came from a SendEvent
                                request */
      Display *display;      /* Display the event was read from */
      Window event;
      Window window;
      int x, y;
} XGravityEvent;
```

The event member is set either to the window that was moved or to its parent, depending on whether *StructureNotify* or *SubstructureNotify* was selected. The window member is set to the child window that was moved. The x and y members are set to the coordinates relative to the new parent window's origin and indicate the position of the upper-left outside corner of the window.

11.10.6 MapNotify Events

The X server can report *MapNotify* events to clients wanting information about which windows are mapped. The X server generates this event type whenever a client application changes the window's state from unmapped to mapped by calling *XMapWindow*, *XMapRaised*, *XMapSubwindows*, *XReparentWindow*, or as a result of save-set processing.

To receive *MapNotify* events, set the *StructureNotifyMask* bit in the event-mask attribute of the window or the *SubstructureNotifyMask* bit in the event-mask attribute of the parent window (in which case, mapping any child generates an event).

The structure for this event type contains:

```
typedef struct {
       int type;               /* MapNotify */
       unsigned long serial;   /* # of last request processed by
                                  server */
       Bool send_event;        /* true if this came from a SendEvent
                                  request */
       Display *display;       /* Display the event was read from */
       Window event;
       Window window;
       Bool override_redirect; /* boolean, is override set... */
} XMapEvent;
```

The event member is set either to the window that was mapped or to its parent, depending on whether *StructureNotify* or *SubstructureNotify* was selected. The window member is set to the window that was mapped. The override_redirect member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the override-redirect attribute is *True*, because these events usually are generated from pop-ups, which override structure control.

11.10.7 MappingNotify Events

The X server reports *MappingNotify* events to all clients. There is no mechanism to express disinterest in this event. The X server generates this event type whenever a client application successfully calls:

- *XSetModifierMapping* to indicate which KeyCodes are to be used as modifiers

- *XChangeKeyboardMapping* to change the keyboard mapping

- *XSetPointerMapping* to set the pointer mapping.

  The structure for this event type contains:

```
        typedef struct {
            int type;                /* MappingNotify */
            unsigned long serial; /* # of last request processed by
                                        server */
            Bool send_event;        /* true if this came from a SendEvent
                                        request */
            Display *display;       /* Display the event was read from */
            Window window;          /* unused */
            int request;            /* one of MappingModifier,
                                        MappingKeyboard, MappingPointer */
            int first_keycode;      /* first keycode */
            int count;              /* defines range of change w.
                                        first_keycode*/
    } XMappingEvent;
```

The request member is set to indicate the kind of mapping change that occurred and can be
MappingModifier, *MappingKeyboard*, *MappingPointer*. If it is *MappingModifier*, the modifier
mapping was changed. If it is *MappingKeyboard*, the keyboard mapping was changed. If it is
MappingPointer, the pointer button mapping was changed. The first_keycode and count
members are set only if the request member was set to *MappingKeyboard*. The number in
first_keycode represents the first number in the range of the altered mapping, and count
represents the number of keycodes altered.

To update the client application's knowledge of the keyboard, you should call
XRefreshKeyboardMapping.

11.10.8 ReparentNotify Events

The X server can report *ReparentNotify* events to clients wanting information about changing a
window's parent. The X server generates this event whenever a client application calls
XReparentWindow and the window is actually reparented.

To receive *ReparentNotify* events, set the *StructureNotifyMask* bit in the event-mask attribute of
the window or the *SubstructureNotifyMask* bit in the event-mask attribute of either the old or the
new parent window (in which case, reparenting any child generates an event).

The structure for this event type contains:

```
    typedef struct {
            int type;                /* ReparentNotify */
            unsigned long serial; /* # of last request processed by
                                        server */
            Bool send_event;        /* true if this came from a SendEvent
                                        request */
            Display *display;       /* Display the event was read from */
            Window event;
            Window window;
            Window parent;
            int x, y;
            Bool override_redirect;
    } XReparentEvent;
```

The event member is set either to the reparented window or to the old or the new parent,
depending on whether *StructureNotify* or *SubstructureNotify* was selected. The window member
is set to the window that was reparented. The parent member is set to the new parent window.
The x and y members are set to the reparented window's coordinates relative to the new parent

window's origin and define the upper-left outer corner of the reparented window. The override_redirect member is set to the override-redirect attribute of the window specified by the window member. Window manager clients normally should ignore this window if the override_redirect member is *True*.

11.10.9 UnmapNotify Events

The X server can report *UnmapNotify* events to clients wanting information about which windows are unmapped. The X server generates this event type whenever a client application changes the window's state from mapped to unmapped.

To receive *UnmapNotify* events, set the *StructureNotifyMask* bit in the event-mask attribute of the window or the *SubstructureNotifyMask* bit in the event-mask attribute of the parent window (in which case, unmapping any child window generates an event).

The structure for this event type contains:

```
typedef struct {
        int type;               /* UnmapNotify */
        unsigned long serial; /* # of last request processed by
                                    server */
        Bool send_event;        /* true if this came from a SendEvent
                                    request */
        Display *display;       /* Display the event was read from */
        Window event;
        Window window;
        Bool from_configure;
} XUnmapEvent;
```

The event member is set either to the unmapped window or to its parent, depending on whether *StructureNotify* or *SubstructureNotify* was selected. This is the window used by the X server to report the event. The window member is set to the window that was unmapped. The from_configure member is set to *True* if the event was generated as a result of a resizing of the window's parent when the window itself had a win_gravity of *UnmapGravity*.

11.10.10 VisibilityNotify Events

The X server can report *VisibilityNotify* events to clients wanting any change in the visibility of the specified window. A region of a window is visible if someone looking at the screen can actually see it. The X server generates this event whenever the visibility changes state. However, this event is never generated for windows whose class is *InputOnly*.

All *VisibilityNotify* events caused by a hierarchy change are generated after any hierarchy event (*UnmapNotify*, *MapNotify*, *ConfigureNotify*, *GravityNotify*, *CirculateNotify*) caused by that change. Any *VisibilityNotify* event on a given window is generated before any *Expose* events on that window, but it is not required that all *VisibilityNotify* events on all windows be generated before all *Expose* events on all windows. The X protocol does not constrain the ordering of *VisibilityNotify* events with respect to *FocusOut*, *EnterNotify* and *LeaveNotify* events.

To receive *VisibilityNotify* events, set the *VisibilityChangeMask* bit in the event-mask attribute of the window.

The structure for this event type contains:

```
typedef struct {
      int type;                /* VisibiltyNotify */
      unsigned long serial; /* # of last request processed by
                                 server */
      Bool send_event;       /* true if this came from a SendEvent
                                 request */
      Display *display;      /* Display the event was read from */
      Window window;
      int state;
} XVisibilityEvent;
```

The window member is set to the window whose visibility state changes. The state member is set to the state of the window's visibility and can be *VisibilityUnobscured*, *VisibilityPartiallyObscured* or *VisibilityFullyObscured*. The X server ignores all of a window's subwindows when determining the visibility state of the window and processes *VisibilityNotify* events according to the following:

- When the window changes state from partially obscured, fully obscured or not viewable to viewable and completely unobscured, the X server generates the event with the state member of the *XVisibilityEvent* structure set to *VisibilityUnobscured*.

- When the window changes state from viewable and completely unobscured or not viewable to viewable and partially obscured, the X server generates the event with the state member of the *XVisibilityEvent* structure set to *VisibilityPartiallyObscured*.

- When the window changes state from viewable and completely unobscured, viewable and partially obscured, or not viewable to viewable and fully obscured, the X server generates the event with the state member of the *XVisibilityEvent* structure set to *VisibilityFullyObscured*.

11.11 Structure Control Events

This section discusses:

- *CirculateRequest* events
- *ConfigureRequest* events
- *MapRequest* events
- *ResizeRequest* events.

11.11.1 CirculateRequest Events

The X server can report *CirculateRequest* events to clients wanting information about when another client initiates a circulate window request on a specified window. The X server generates this event type whenever a client initiates a circulate window request on a window and a subwindow actually needs to be restacked. The client initiates a circulate window request on the window by calling *XCirculateSubwindows*, *XCirculateSubwindowsUp* or *XCirculateSubwindowsDown*.

To receive *CirculateRequest* events, set the *SubstructureRedirectMask* in the event-mask attribute of the window. Then, in the future, the circulate window request for the specified window is not executed, and thus, any subwindow's position in the stack is not changed. For example, suppose a client application calls *XCirculateSubwindowsUp* to raise a subwindow to the top of the stack. If you had selected *SubstructureRedirectMask* on the window, the X server reports to you a *CirculateRequest* event and does not raise the subwindow to the top of the stack.

The structure for this event type contains:

```
typedef struct {
      int type;               /* CirculateRequest */
      unsigned long serial; /* # of last request processed by
                                 server */
      Bool send_event;        /* true if this came from a SendEvent
                                 request */
      Display *display;       /* Display the event was read from */
      Window parent;
      Window window;
      int place;              /* PlaceOnTop, PlaceOnBottom */
} XCirculateRequestEvent;
```

The parent member is set to the parent window. The window member is set to the subwindow to be restacked. The place member is set to what the new position in the stacking order should be and is either *PlaceOnTop* or *PlaceOnBottom*. If it is *PlaceOnTop*, the subwindow should be on top of all siblings. If it is *PlaceOnBottom*, the subwindow should be below all siblings.

11.11.2 ConfigureRequest Events

The X server can report *ConfigureRequest* events to clients wanting information about when a different client initiates a configure window request on any child of a specified window. The configure window request attempts to reconfigure a window's size, position, border and stacking order. The X server generates this event whenever a different client initiates a configure window request on a window by calling *XConfigureWindow*, *XLowerWindow*, *XRaiseWindow*, *XMapRaised*, *XMoveResizeWindow*, *XMoveWindow*, *XResizeWindow*, *XRestackWindows* or *XSetWindowBorderWidth*.

To receive *ConfigureRequest* events, set the *SubstructureRedirectMask* bit in the event-mask attribute of the window. *ConfigureRequest* events are generated when a *ConfigureWindow* protocol request is issued on a child window by another client. For example, suppose a client application calls *XLowerWindow* to lower a window. If you had selected *SubstructureRedirectMask* on the parent window and if the override-redirect attribute of the window is set to *False*, the X server reports a *ConfigureRequest* event to you and does not lower the specified window.

The structure for this event type contains:

```
typedef struct {
        int type;               /* ConfigureRequest */
        unsigned long serial; /* # of last request processed by
                                      server */
        Bool send_event;        /* true if this came from a SendEvent
                                      request */
        Display *display;       /* Display the event was read from */
        Window parent;
        Window window;
        int x, y;
        int width, height;
        int border_width;
        Window above;
        int detail;             /* Above, Below, TopIf, BottomIf,
                                      Opposite */
        unsigned long value_mask;
} XConfigureRequestEvent;
```

The parent member is set to the parent window. The window member is set to the window whose size, position, border width and/or stacking order is to be reconfigured. The value_mask member indicates which components were specified in the *ConfigureWindow* protocol request. The corresponding values are reported as given in the request. The remaining values are filled in from the current geometry of the window, except in the case of above (sibling) and detail (stack-mode), which are reported as *Above* and *None*, respectively, if they are not given in the request.

11.11.3 MapRequest Events

The X server can report *MapRequest* events to clients wanting information about a different client's desire to map windows. A window is considered mapped when a map window request completes. The X server generates this event whenever a different client initiates a map window request on an unmapped window whose override_redirect member is set to *False*. Clients initiate map window requests by calling *XMapWindow*, *XMapRaised* or *XMapSubwindows*.

To receive *MapRequest* events, set the *SubstructureRedirectMask* bit in the event-mask attribute of the window. This means another client's attempts to map a child window by calling one of the map window request functions is intercepted, and you are sent a *MapRequest* instead. For example, suppose a client application calls *XMapWindow* to map a window. If you (usually a window manager) had selected *SubstructureRedirectMask* on the parent window and if the override-redirect attribute of the window is set to *False*, the X server reports a *MapRequest* event to you and does not map the specified window. Thus, this event gives your window manager client the ability to control the placement of subwindows.

The structure for this event type contains:

```
typedef struct {
    int type;               /* MapRequest */
    unsigned long serial; /* # of last request processed by
                              server */
    Bool send_event;        /* true if this came from a SendEvent
                              request */
    Display *display;       /* Display the event was read from */
    Window parent;
    Window window;
} XMapRequestEvent;
```

The parent member is set to the parent window. The window member is set to the window to be mapped.

11.11.4 ResizeRequest Events

The X server can report *ResizeRequest* events to clients wanting information about another client's attempts to change the size of a window. The X server generates this event whenever some other client attempts to change the size of the specified window by calling *XConfigureWindow*, *XResizeWindow* or *XMoveResizeWindow*.

To receive *ResizeRequest* events, set the *ResizeRedirect* bit in the event-mask attribute of the window. Any attempts to change the size by other clients are then redirected.

The structure for this event type contains:

```
typedef struct {
    int type;               /* ResizeRequest */
    unsigned long serial; /* # of last request processed by
                              server */
    Bool send_event;        /* true if this came from a SendEvent
                              request */
    Display *display;       /* Display the event was read from */
    Window window;
    int width, height;
} XResizeRequestEvent;
```

The window member is set to the window whose size another client attempted to change. The width and height members are set to the inside size of the window, excluding the border.

11.12 Colormap State Change Events

The X server can report *ColormapNotify* events to clients wanting information about when the colormap changes and when a colormap is installed or uninstalled. The X server generates this event type whenever a client application:

- changes the colormap member of the *XSetWindowAttributes* structure by calling *XChangeWindowAttributes*, *XFreeColormap* or *XSetWindowColormap*

- installs or uninstalls the colormap by calling *XInstallColormap* or *XUninstallColormap*.

To receive *ColormapNotify* events, set the *ColormapChangeMask* bit in the event-mask attribute of the window.

The structure for this event type contains:

```
typedef struct {
      int type;               /* ColormapNotify */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;        /* true if this came from a SendEvent
                                request */
      Display *display;       /* Display the event was read from */
      Window window;
      Colormap colormap;      /* colormap or None */
      Bool new;
      int state;              /* ColormapInstalled,
                                ColormapUninstalled */
} XColormapEvent;
```

The window member is set to the window whose associated colormap is changed, installed or uninstalled. For a colormap that is changed, installed or uninstalled, the colormap member is set to the colormap associated with the window. For a colormap that is changed by a call to *XFreeColormap*, the colormap member is set to *None*. The new member is set to indicate whether the colormap for the specified window was changed or installed or uninstalled and can be *True* or *False*. If it is *True*, the colormap was changed. If it is *False*, the colormap was installed or uninstalled. The state member is always set to indicate whether the colormap is installed or uninstalled and can be *ColormapInstalled* or *ColormapUninstalled*.

11.13   Client Communication Events

This section discusses:

- *ClientMessage* events

- *PropertyNotify* events

- *SelectionClear* events

- *SelectionNotify* events

- *SelectionRequest* events.

11.13.1   ClientMessage Events

The X server generates *ClientMessage* events only when a client calls the function *XSendEvent*.

The structure for this event type contains:

```
typedef struct {
      int type;               /* ClientMessage */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;        /* true if this came from a SendEvent
                                request */
      Display *display;       /* Display the event was read from */
      Window window;
      Atom message_type;
      int format;
      union {
            char b[20];
            short s[10];
            long l[5];
              } data;
} XClientMessageEvent;
```

The message_type member is set to an atom that indicates how the data should be interpreted by the receiving client. The format member is set to 8, 16 or 32 and specifies whether the data should be viewed as a list of bytes, shorts, or longs. The data member is a union that contains the members b, s and l. The b, s and l members represent data of 20 8-bit values, 10 16-bit values and 5 32-bit values. Particular message types might not make use of all these values. The X server places no interpretation on the values in the window, message_type, or data members.

11.13.2   PropertyNotify Events

The X server can report *PropertyNotify* events to clients wanting information about property changes for a specified window.

To receive *PropertyNotify* events, set the *PropertyChangeMask* bit in the event-mask attribute of the window.

The structure for this event type contains:

```
typedef struct {
      int type;              /* PropertyNotify */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;       /* true if this came from a SendEvent
                                request */
      Display *display;      /* Display the event was read from */
      Window window;
      Atom atom;
      Time time;
      int state;             /* PropertyNewValue or PropertyDelete */
} XPropertyEvent;
```

The window member is set to the window whose associated property was changed. The atom member is set to the property's atom and indicates which property was changed or desired. The time member is set to the server time when the property was changed. The state member is set to indicate whether the property was changed to a new value or deleted and can be *PropertyNewValue* or *PropertyDelete*. The state member is set to *PropertyNewValue* when a property of the window is changed using *XChangeProperty* or *XRotateWindowProperties* (even when adding zero-length data using *XChangeProperty*) and when replacing all or part of a property with identical data using *XChangeProperty* or *XRotateWindowProperties*. The state member is set to *PropertyDelete* when a property of the window is deleted using *XDeleteProperty* or, if the delete argument is *True*, *XGetWindowProperty*.

11.13.3 SelectionClear Events

The X server reports *SelectionClear* events to the client losing ownership of a selection. The X server generates this event type when another client asserts ownership of the selection by calling *XSetSelectionOwner*.

The structure for this event type contains:

```
typedef struct {
      int type;              /* SelectionClear */
      unsigned long serial; /* # of last request processed by
                                server */
      Bool send_event;       /* true if this came from a SendEvent
                                request */
      Display *display;      /* Display the event was read from */
      Window window;
      Atom selection;
      Time time;
} XSelectionClearEvent;
```

The selection member is set to the selection atom. The time member is set to the last change time recorded for the selection. The window member is the window that was specified by the current owner (the owner losing the selection) in its *XSetSelectionOwner* call.

11.13.4 SelectionRequest Events

The X server reports *SelectionRequest* events to the owner of a selection. The X server generates this event whenever a client requests a selection conversion by calling *XConvertSelection* for the owned selection.

The structure for this event type contains:

```
typedef struct {
        int type;               /* SelectionRequest */
        unsigned long serial; /* # of last request processed by
                                  server */
        Bool send_event;      /* true if this came from a SendEvent
                                  request */
        Display *display;     /* Display the event was read from */
        Window owner;
        Window requestor;
        Atom selection;
        Atom target;
        Atom property;
        Time time;
} XSelectionRequestEvent;
```

The owner member is set to the window that was specified by the current owner in its *XSetSelectionOwner* call. The requestor member is set to the window requesting the selection. The selection member is set to the atom that names the selection. For example, PRIMARY is used to indicate the primary selection. The target member is set to the atom that indicates the type the selection is desired in. The property member can be a property name or *None*. The time member is set to the timestamp or *CurrentTime* value from the *ConvertSelection* request.

The owner should convert the selection based on the specified target type and send a *SelectionNotify* event back to the requestor. A complete specification for using selections is given in the **ICCCM** specification.

11.13.5 SelectionNotify Events

This event is generated by the X server in response to a *ConvertSelection* protocol request when there is no owner for the selection. When there is an owner, it should be generated by the owner of the selection by using *XSendEvent*. The owner of a selection should send this event to a requestor when a selection has been converted and stored as a property or when a selection conversion could not be performed (which is indicated by setting the property member to *None*).

If *None* is specified as the property in the *ConvertSelection* protocol request, the owner should choose a property name, store the result as that property on the requestor window, and then send a *SelectionNotify* giving that actual property name.

The structure for this event type contains:

```
typedef struct {
      int type;               /* SelectionNotify */
      unsigned long serial; /* # of last request processed by
                                   server */
      Bool send_event;        /* true if this came from a SendEvent
                                   request */
      Display *display;       /* Display the event was read from */
      Window requestor;
      Atom selection;
      Atom target;
      Atom property;          /* atom or None */
      Time time;
} XSelectionEvent;
```

The requestor member is set to the window associated with the requestor of the selection. The selection member is set to the atom that indicates the selection. For example, PRIMARY is used for the primary selection. The target member is set to the atom that indicates the converted type. For example, PIXMAP is used for a pixmap. The property member is set to the atom that indicates which property the result was stored on. If the conversion failed, the property member is set to *None*. The time member is set to the time the conversion took place and can be a timestamp or *CurrentTime*.

Event Handling Functions

This chapter discusses the Xlib functions you can use to:

- select events

- handle the output buffer and the event queue

- select events from the event queue

- send and get events

- handle protocol errors.

Note: Some toolkits use their own event-handling functions and do not allow you to interchange these event-handling functions with those in Xlib. For further information, see the documentation supplied with the toolkit.

Most applications simply are event loops: they wait for an event, decide what to do with it, execute some amount of code that results in changes to the display, and then wait for the next event.

12.1 Selecting Events

There are two ways to select the events you want reported to your client application. One way is to set the event_mask member of the *XSetWindowAttributes* structure when you call *XCreateWindow* and *XChangeWindowAttributes*. Another way is to use *XSelectInput*.

```
XSelectInput(display, w, event_mask)
      Display *display;
      Window w;
      long event_mask;
```

display   Specifies the connection to the X server.

w   Specifies the window whose events you are interested in.

event_mask   Specifies the event mask.

The *XSelectInput* function requests that the X server report the events associated with the specified event mask. Initially, X will not report any of these events. Events are reported relative to a window. If a window is not interested in a device event, it usually propagates to the closest ancestor that is interested, unless the do_not_propagate mask prohibits it.

Setting the event-mask attribute of a window overrides any previous call for the same window but not for other clients. Multiple clients can select for the same events on the same window with the following restrictions:

- Multiple clients can select events on the same window because their event masks are disjoint. When the X server generates an event, it reports it to all interested clients.

- Only one client at a time can select *CirculateRequest*, *ConfigureRequest* or *MapRequest* events, which are associated with the event mask *SubstructureRedirectMask*.

- Only one client at a time can select a *ResizeRequest* event, which is associated with the event mask *ResizeRedirectMask*.

- Only one client at a time can select a *ButtonPress* event, which is associated with the event mask *ButtonPressMask*.

The server reports the event to all interested clients.

XSelectInput can generate a *BadWindow* error.

12.2   Handling the Output Buffer

The output buffer is an area used by Xlib to store requests. The functions described in this section flush the output buffer if the function would block or not return an event. That is, all requests residing in the output buffer that have not yet been sent are transmitted to the X server. These functions differ in the additional tasks they might perform.

To flush the output buffer, use *XFlush*.

```
XFlush(display)
       Display *display;
```

display   Specifies the connection to the X server.

The *XFlush* function flushes the output buffer. Most client applications need not use this function because the output buffer is automatically flushed as needed by calls to *XPending*, *XNextEvent* and *XWindowEvent*. Events generated by the server may be enqueued into the library's event queue.

To flush the output buffer and then wait until all requests have been processed, use *XSync*.

```
XSync(display, discard)
       Display *display;
       Bool discard;
```

display   Specifies the connection to the X server.

discard   Specifies a Boolean value that indicates whether *XSync* discards all events on the event queue.

The *XSync* function flushes the output buffer and then waits until all requests have been received and processed by the X server. Any errors generated must be handled by the error handler. For each protocol error received by Xlib, *XSync* calls the client application's error handling routine (see Section 12.8.2). Any events generated by the server are enqueued into the library's event queue.

Finally, if you passed *False*, *XSync* does not discard the events in the queue. If you passed *True*, *XSync* discards all events in the queue, including those events that were on the queue before *XSync* was called. Client applications seldom need to call *XSync*.

12.3   Event Queue Management

Xlib maintains an event queue. However, the operating system also may be buffering data in its network connection that is not yet read into the event queue.

To check the number of events in the event queue, use *XEventsQueued*.

```
int XEventsQueued(display, mode)
    Display *display;
    int mode;
```

display   Specifies the connection to the X server.

mode   Specifies the mode. You can pass *QueuedAlready*, *QueuedAfterFlush* or *QueuedAfterReading*.

If mode is *QueuedAlready*, *XEventsQueued* returns the number of events already in the event queue (and never performs a system call). If mode is *QueuedAfterFlush*, *XEventsQueued* returns the number of events already in the queue if the number is non-zero. If there are no events in the queue, *XEventsQueued* flushes the output buffer, attempts to read more events out of the application's connection, and returns the number read. If mode is *QueuedAfterReading*, *XEventsQueued* returns the number of events already in the queue if the number is non-zero. If there are no events in the queue, *XEventsQueued* attempts to read more events out of the application's connection without flushing the output buffer and returns the number read.

XEventsQueued always returns immediately without I/O if there are events already in the queue. *XEventsQueued* with mode *QueuedAfterFlush* is identical in behaviour to *XPending*. *XEventsQueued* with mode *QueuedAlready* is identical to the *XQLength* function.

To return the number of events that are pending, use *XPending*.

```
int XPending(display)
    Display *display;
```

display   Specifies the connection to the X server.

The *XPending* function returns the number of events that have been received from the X server but have not been removed from the event queue. *XPending* is identical to *XEventsQueued* with the mode *QueuedAfterFlush* specified.

12.4 Manipulating the Event Queue

Xlib provides functions that let you manipulate the event queue. This section discusses how to:

- obtain events, in order, and remove them from the queue
- peek at events in the queue without removing them
- obtain events that match the event mask or the arbitrary predicate procedures that you provide.

12.4.1 Returning the Next Event

To get the next event and remove it from the queue, use *XNextEvent*.

```
XNextEvent(display, event_return)
      Display *display;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

event_return   Returns the next event in the queue.

The *XNextEvent* function copies the first event from the event queue into the specified *XEvent* structure and then removes it from the queue. If the event queue is empty, *XNextEvent* flushes the output buffer and blocks until an event is received.

To peek at the event queue, use *XPeekEvent*.

```
XPeekEvent(display, event_return)
      Display *display;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

event_return   Returns a copy of the matched event's associated structure.

The *XPeekEvent* function returns the first event from the event queue, but it does not remove the event from the queue. If the queue is empty, *XPeekEvent* flushes the output buffer and blocks until an event is received. It then copies the event into the client-supplied *XEvent* structure without removing it from the event queue.

12.4.2 Selecting Events Using a Predicate Procedure

Each of the functions discussed in this section requires you to pass a predicate procedure that determines if an event matches what you want. Your predicate procedure must decide only if the event is useful and must not call Xlib functions. In particular, a predicate is called from inside the event routine, which must lock data structures so that the event queue is consistent in a multi-threaded environment.

The predicate procedure and its associated arguments are:

```
Bool (*predicate)(display, event, arg)
      Display *display;
      XEvent *event;
      XPointer arg;
```

display   Specifies the connection to the X server.

event   Specifies the *XEvent* structure.

arg   Specifies the argument passed in from the *XIfEvent*, *XCheckIfEvent* or
                    XPeekIfEvent function.

The predicate procedure is called once for each event in the queue until it finds a match. After
finding a match, the predicate procedure must return *True*. If it did not find a match, it must
return *False*.

To check the event queue for a matching event and, if found, remove the event from the queue,
use *XIfEvent*.

```
XIfEvent(display, event_return, predicate, arg)
      Display *display;
      XEvent *event_return;
      Bool (*predicate)();
      XPointer arg;
```

display   Specifies the connection to the X server.

event_return   Returns the matched event's associated structure.

predicate   Specifies the procedure that is to be called to determine if the next event in the
                    queue matches what you want.

arg   Specifies the user-supplied argument that will be passed to the predicate
                    procedure.

The *XIfEvent* function completes only when the specified predicate procedure returns *True* for an
event, which indicates an event in the queue matches. *XIfEvent* flushes the output buffer if it
blocks waiting for additional events. *XIfEvent* removes the matching event from the queue and
copies the structure into the client-supplied *XEvent* structure.

To check the event queue for a matching event without blocking, use *XCheckIfEvent*.

```
Bool XCheckIfEvent(display, event_return, predicate,
          arg)
      Display *display;
      XEvent *event_return;
      Bool (*predicate)();
      XPointer arg;
```

display   Specifies the connection to the X server.

event_return   Returns a copy of the matched event's associated structure.

predicate   Specifies the procedure that is to be called to determine if the next event in the
                    queue matches what you want.

arg   Specifies the user-supplied argument that will be passed to the predicate
                    procedure.

When the predicate procedure finds a match, *XCheckIfEvent* copies the matched event into the
client-supplied *XEvent* structure and returns *True*. (This event is removed from the queue.) If
the predicate procedure finds no match, *XCheckIfEvent* returns *False*, and the output buffer will
have been flushed. All earlier events stored in the queue are not discarded.

To check the event queue for a matching event without removing the event from the queue, use
XPeekIfEvent.

```
XPeekIfEvent(display, event_return, predicate, arg)
      Display *display;
      XEvent *event_return;
      Bool (*predicate)();
      XPointer arg;
```

display   Specifies the connection to the X server.

event_return   Returns a copy of the matched event's associated structure.

predicate   Specifies the procedure that is to be called to determine if the next event in the
                     queue matches what you want.

arg   Specifies the user-supplied argument that will be passed to the predicate
                     procedure.

The *XPeekIfEvent* function returns only when the specified predicate procedure returns *True* for
an event. After the predicate procedure finds a match, *XPeekIfEvent* copies the matched event
into the client-supplied *XEvent* structure without removing the event from the queue.
XPeekIfEvent flushes the output buffer if it blocks waiting for additional events.

12.4.3   Selecting Events Using a Window or Event Mask

The functions discussed in this section let you select events by window or event types, allowing
you to process events out of order.

To remove the next event that matches both a window and an event mask, use *XWindowEvent*.

```
XWindowEvent(display, w, event_mask, event_return)
      Display *display;
      Window w;
      long event_mask;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

w   Specifies the window whose events you are interested in.

event_mask   Specifies the event mask.

event_return   Returns the matched event's associated structure.

The *XWindowEvent* function searches the event queue for an event that matches both the
specified window and event mask. When it finds a match, *XWindowEvent* removes that event
from the queue and copies it into the specified *XEvent* structure. The other events stored in the
queue are not discarded. If a matching event is not in the queue, *XWindowEvent* flushes the
output buffer and blocks until one is received.

To remove the next event that matches both a window and an event mask (if any), use
XCheckWindowEvent. This function is similar to *XWindowEvent* except that it never blocks and it
returns a *Bool* indicating if the event was returned.

```
Bool XCheckWindowEvent(display, w, event_mask,
            event_return)
      Display *display;
      Window w;
      long event_mask;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

w   Specifies the window whose events you are interested in.

event_mask   Specifies the event mask.

event_return   Returns the matched event's associated structure.

The *XCheckWindowEvent* function searches the event queue and then the events available on the server connection for the first event that matches the specified window and event mask. If it finds a match, *XCheckWindowEvent* removes that event, copies it into the specified *XEvent* structure, and returns *True*. The other events stored in the queue are not discarded. If the event you requested is not available, *XCheckWindowEvent* returns *False*, and the output buffer will have been flushed.

To remove the next event that matches an event mask, use *XMaskEvent*.

```
XMaskEvent(display, event_mask, event_return)
      Display *display;
      long event_mask;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

event_mask   Specifies the event mask.

event_return   Returns the matched event's associated structure.

The *XMaskEvent* function searches the event queue for the events associated with the specified mask. When it finds a match, *XMaskEvent* removes that event and copies it into the specified *XEvent* structure. The other events stored in the queue are not discarded. If the event you requested is not in the queue, *XMaskEvent* flushes the output buffer and blocks until one is received.

To return and remove the next event that matches an event mask (if any), use *XCheckMaskEvent*. This function is similar to *XMaskEvent* except that it never blocks and it returns a *Bool* indicating if the event was returned.

```
Bool XCheckMaskEvent(display, event_mask, event_return)
      Display *display;
      long event_mask;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

event_mask   Specifies the event mask.

event_return   Returns the matched event's associated structure.

The *XCheckMaskEvent* function searches the event queue and then any events available on the server connection for the first event that matches the specified mask. If it finds a match, *XCheckMaskEvent* removes that event, copies it into the specified *XEvent* structure, and returns *True*. The other events stored in the queue are not discarded. If the event you requested is not available, *XCheckMaskEvent* returns *False*, and the output buffer will have been flushed.

To return and remove the next event in the queue that matches an event type, use *XCheckTypedEvent*.

```
Bool XCheckTypedEvent(display, event_type, event_return)
      Display *display;
      int event_type;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

event_type   Specifies the event type to be compared.

event_return   Returns the matched event's associated structure.

The *XCheckTypedEvent* function searches the event queue and then any events available on the server connection for the first event that matches the specified type. If it finds a match, *XCheckTypedEvent* removes that event, copies it into the specified *XEvent* structure, and returns *True*. The other events in the queue are not discarded. If the event is not available, *XCheckTypedEvent* returns *False*, and the output buffer will have been flushed.

To return and remove the next event in the queue that matches an event type and a window, use *XCheckTypedWindowEvent*.

```
Bool XCheckTypedWindowEvent(display, w, event_type,
          event_return)
      Display *display;
      Window w;
      int event_type;
      XEvent *event_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

event_type   Specifies the event type to be compared.

event_return   Returns the matched event's associated structure.

The *XCheckTypedWindowEvent* function searches the event queue and then any events available on the server connection for the first event that matches the specified type and window. If it finds a match, *XCheckTypedWindowEvent* removes the event from the queue, copies it into the specified *XEvent* structure, and returns *True*. The other events in the queue are not discarded. If the event is not available, *XCheckTypedWindowEvent* returns *False*, and the output buffer will have been flushed.

12.5   Putting an Event Back into the Queue

To push an event back into the event queue, use *XPutBackEvent*.

```
XPutBackEvent(display, event)
      Display *display;
      XEvent *event;
```

display   Specifies the connection to the X server.

event   Specifies the event.

The *XPutBackEvent* function pushes an event back onto the head of the display's event queue by copying the event into the queue. This can be useful if you read an event and then decide that you would rather deal with it later. There is no limit to the number of times in succession that you can call *XPutBackEvent*.

12.6   Sending Events to Other Applications

To send an event to a specified window, use *XSendEvent*. This function is often used in selection processing. For example, the owner of a selection should use *XSendEvent* to send a *SelectionNotify* event to a requestor when a selection has been converted and stored as a property.

```
Status XSendEvent(display, w, propagate, event_mask,
        event_send)
    Display *display;
    Window w;
    Bool propagate;
    long event_mask;
    XEvent *event_send;
```

display   Specifies the connection to the X server.

w   Specifies the window the event is to be sent to, *PointerWindow*, or *InputFocus*.

propagate   Specifies a Boolean value.

event_mask   Specifies the event mask.

event_send   Specifies the event that is to be sent.

The *XSendEvent* function identifies the destination window, determines which clients should receive the specified events, and ignores any active grabs. This function requires you to pass an event mask. For a discussion of the valid event mask names, see Section 11.3. This function uses the w argument to identify the destination window as follows:

- If w is *PointerWindow*, the destination window is the window that contains the pointer.

- If w is *InputFocus* and if the focus window contains the pointer, the destination window is the window that contains the pointer; otherwise, the destination window is the focus window.

To determine which clients should receive the specified events, *XSendEvent* uses the propagate argument as follows:

- If event_mask is the empty set, the event is sent to the client that created the destination window. If that client no longer exists, no event is sent.

- If propagate is *False*, the event is sent to every client selecting on destination any of the event types in the event_mask argument.

- If propagate is *True* and no clients have selected on destination any of the event types in event-mask, the destination is replaced with the closest ancestor of destination for which some client has selected a type in event-mask and for which no intervening window has that type in its do-not-propagate-mask. If no such window exists or if the window is an ancestor of the focus window and *InputFocus* was originally specified as the destination, the event is not sent to any clients. Otherwise, the event is reported to every client selecting on the final destination any of the types specified in event_mask.

The event in the *XEvent* structure must be one of the core events or one of the events defined by an extension (or a *BadValue* error results) so that the X server can correctly byte-swap the contents as necessary. The contents of the event are otherwise unaltered and unchecked by the X server except to force send_event to *True* in the forwarded event and to set the serial number in the event correctly.

XSendEvent returns zero if the conversion to wire protocol format failed and returns non-zero otherwise.

XSendEvent can generate *BadValue* and *BadWindow* errors.

12.7 Getting Pointer Motion History

Some X server implementations will maintain a more complete history of pointer motion than is reported by event notification. The pointer position at each pointer hardware interrupt may be stored in a buffer for later retrieval. This buffer is called the motion history buffer. For example, a few applications, such as paint programs, want to have a precise history of where the pointer travelled. However, this historical information is highly excessive for most applications.

To determine the approximate maximum number of elements in the motion buffer, use *XDisplayMotionBufferSize*.

```
unsigned long XDisplayMotionBufferSize(display)
        Display *display;
```

display   Specifies the connection to the X server.

The server may retain the recent history of the pointer motion and do so to a finer granularity than is reported by *MotionNotify* events. The *XGetMotionEvents* function makes this history available.

To get the motion history for a specified window and time, use *XGetMotionEvents*.

```
XTimeCoord *XGetMotionEvents(display, w, start, stop,
        nevents_return)
    Display *display;
    Window w;
    Time start, stop;
    int *nevents_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

start
stop   Specify the time interval in which the events are returned from the motion history buffer. You can pass a timestamp or *CurrentTime*.

nevents_return   Returns the number of events from the motion history buffer.

The *XGetMotionEvents* function returns all events in the motion history buffer that fall between the specified start and stop times, inclusive, and that have coordinates that lie within the specified window (including its borders) at its present placement. If the server does not support motion history, or if the start time is later than the stop time, or if the start time is in the future, no events are returned, and *XGetMotionEvents* returns NULL. If the stop time is in the future, it is equivalent to specifying *CurrentTime*. The return type for this function is a structure defined as follows:

```
typedef struct {
        Time time;
        short x, y;
} XTimeCoord;
```

The time member is set to the time, in milliseconds. The x and y members are set to the coordinates of the pointer and are reported relative to the origin of the specified window. To free the data returned from this call, use *XFree*.

XGetMotionEvents can generate a *BadWindow* error.

12.8   Handling Protocol Errors

Xlib provides functions that you can use to enable or disable synchronization and to use the default error handlers.

12.8.1   Enabling or Disabling Synchronization

When debugging X applications, it often is very convenient to require Xlib to behave synchronously so that errors are reported as they occur. The following function lets you disable or enable synchronous behaviour. Note that graphics may occur 30 or more times more slowly when synchronization is enabled. On POSIX-conformant systems, there is also a global variable *_Xdebug* that, if set to non-zero before starting a program under a debugger, will force synchronous library behaviour.

After completing their work, all Xlib functions that generate protocol requests call what is known as an after function. *XSetAfterFunction* sets which function is to be called.

```
int (*XSetAfterFunction(display, procedure))()
      Display *display;
      int (*procedure)();
```

display   Specifies the connection to the X server.

procedure   Specifies the function to be called.

The specified procedure is called with only a display pointer. *XSetAfterFunction* returns the previous after function.

To enable or disable synchronization, use *XSynchronize*.

```
int (*XSynchronize(display, onoff))()
      Display *display;
      Bool onoff;
```

display   Specifies the connection to the X server.

onoff   Specifies a Boolean value that indicates whether to enable or disable synchronization.

The *XSynchronize* function returns the previous after function. If onoff is *True*, *XSynchronize* turns on synchronous behaviour. If onoff is *False*, *XSynchronize* turns off synchronous behaviour.

12.8.2   Using the Default Error Handlers

There are two default error handlers in Xlib: one to handle typically fatal conditions (for example, the connection to a display server dying because a machine crashed) and one to handle protocol errors from the X server. These error handlers can be changed to user-supplied routines if you prefer your own error handling and can be changed as often as you like. If either function is passed a NULL pointer, it will reinvoke the default handler. The action of the default handlers is to print an explanatory message and exit.

To set the error handler, use *XSetErrorHandler*.

```
int (*XSetErrorHandler(handler))()
      int (*handler)(Display *, XErrorEvent *)
```

handler   Specifies the program's supplied error handler.

Xlib generally calls the program's supplied error handler whenever an error is received. It is not called on *BadName* errors from *OpenFont*, *LookupColor* or *AllocNamedColor* protocol requests or on *BadFont* errors from a *QueryFont* protocol request. These errors generally are reflected back to the program through the procedural interface. Because this condition is not assumed to be fatal, it is acceptable for your error handler to return. However, the error handler should not call any functions (directly or indirectly) on the display that will generate protocol requests or that will look for input events. The previous error handler is returned.

The *XErrorEvent* structure contains:

```
typedef struct {
int type;
    Display *display;          /* Display the event was read from */
    unsigned long serial;      /* serial number of failed request */
    unsigned char error_code;  /* error code of failed request */
    unsigned char request_code; /* Major op-code of failed request */
    unsigned char minor_code;  /* Minor op-code of failed request */
    XID resourceid;            /* resource id */
} XErrorEvent;
```

The serial member is the number of requests, starting from one, sent over the network connection since it was opened. It is the number that was the value of *NextRequest* immediately before the failing call was made. The request_code member is a protocol request of the procedure that failed, as defined in **<X11/Xproto.h>**. The following error codes can be returned by the functions described in this chapter:

Error Code	Description
BadAccess	A client attempts to grab a key/button combination already grabbed by another client.
	A client attempts to free a colormap entry that it had not already allocated, or to free an entry in a colormap that was created with all entries writable.
	A client attempts to store into a read-only or unallocated colormap entry.
	A client attempts to modify the access control list from other than the local (or otherwise authorized) host.
	A client attempts to select an event type that another client has already selected.
BadAlloc	The server fails to allocate the requested resource. Note that the explicit listing of *BadAlloc* errors in requests only covers allocation errors at a very coarse level and is not intended to (nor can it in practice hope to) cover all cases of a server running out of allocation space in the middle of service. The semantics when a server runs out of allocation space are left unspecified, but a server may generate a *BadAlloc* error on any request for this reason, and clients should be prepared to receive such errors and handle or discard them.

Error Code	Description
BadAtom	A value for an atom argument does not name a defined atom.
BadColor	A value for a colormap argument does not name a defined colormap.
BadCursor	A value for a cursor argument does not name a defined cursor.
BadDrawable	A value for a drawable argument does not name a defined window or pixmap.
BadFont	A value for a font argument does not name a defined font (or, in some cases, *GContext*).
BadGC	A value for a *GContext* argument does not name a defined *GContext*.
BadIDChoice	The value chosen for a resource identifier either is not included in the range assigned to the client or is already in use. Under normal circumstances, this cannot occur and should be considered a server or Xlib error.
BadImplementation	The server does not implement some aspect of the request. A server that generates this error for a core request is deficient. As such, this error is not listed for any of the requests, but clients should be prepared to receive such errors and handle or discard them.
BadLength	The length of a request is shorter or longer than that required to contain the arguments. This is an internal Xlib or server error.
	The length of a request exceeds the maximum length accepted by the server.
BadMatch	In a graphics request, the root and depth of the graphics context does not match that of the drawable.
	An *InputOnly* window is used as a drawable.
	Some argument or pair of arguments has the correct type and range, but it fails to match in some other way required by the request.
	An *InputOnly* window lacks this attribute.
BadName	A font or color of the specified name does not exist.
BadPixmap	A value for a pixmap argument does not name a defined pixmap.
BadRequest	The major or minor opcode does not specify a valid request. This usually is an Xlib or server error.
BadValue	Some numeric value falls outside of the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives typically can generate this error (due to the encoding).
BadWindow	A value for a window argument does not name a defined window.

Note: The *BadAtom*, *BadColor*, *BadCursor*, *BadDrawable*, *BadFont*, *BadGC*, *BadPixmap* and *BadWindow* errors are also used when the argument type is extended by a set of fixed alternatives.

To obtain textual descriptions of the specified error code, use *XGetErrorText*.

```
XGetErrorText(display, code, buffer_return, length)
      Display *display;
      int code;
      char *buffer_return;
      int length;
```

display   Specifies the connection to the X server.

code   Specifies the error code for which you want to obtain a description.

buffer_return   Returns the error description.

length   Specifies the size of the buffer.

The *XGetErrorText* function copies a null-terminated string describing the specified error code into the specified buffer. The returned text is in the encoding of the current locale. It is recommended that you use this function to obtain an error description because extensions to Xlib may define their own error codes and error strings.

To obtain error messages from the error database, use *XGetErrorDatabaseText*.

```
XGetErrorDatabaseText(display, name, message,
          default_string, buffer_return, length)
      Display *display;
      char *name, *message;
      char *default_string;
      char *buffer_return;
      int length;
```

display   Specifies the connection to the X server.

name   Specifies the name of the application.

message   Specifies the type of the error message.

default_string   Specifies the default error message if none is found in the database.

buffer_return   Returns the error description.

length   Specifies the size of the buffer.

The *XGetErrorDatabaseText* function returns a null-terminated message (or the default message) from the error message database. Xlib uses this function internally to look up its error messages. The default_string is assumed to be in the encoding of the current locale. The buffer_return text is in the encoding of the current locale.

The name argument should generally be the name of your application. The message argument should indicate which type of error message you want. If the name and message are not in the Host Portable Character Encoding the result is implementation-dependent. Xlib uses three predefined ''application names'' to report errors (upper case and lower case matter):

XProtoError   The protocol error number is used as a string for the message argument.

XlibMessage   These are the message strings that are used internally by the library.

XRequest   For a core protocol request, the major request protocol number is used for the message argument. For an extension request, the extension name (as given by *InitExtension*) followed by a period (.) and the minor request protocol number is used for the message argument. If no string is found in the error database, the default_string is returned to the buffer argument.

To report an error to the user when the requested display does not exist, use *XDisplayName*.

```
char *XDisplayName(string)
      char *string;
```

string   Specifies the character string.

The *XDisplayName* function returns the name of the display that *XOpenDisplay* would attempt to use. If a NULL string is specified, *XDisplayName* looks in the environment for the display and returns the display name that *XOpenDisplay* would attempt to use. This makes it easier to report to the user precisely which display the program attempted to open when the initial connection attempt failed.

To handle fatal I/O errors, use *XSetIOErrorHandler*.

```
int (*XSetIOErrorHandler(handler))()
      int (*handler)(Display *);
```

handler   Specifies the program's supplied error handler.

The *XSetIOErrorHandler* sets the fatal I/O error handler. Xlib calls the program's supplied error handler if any sort of system call error occurs (for example, the connection to the server was lost). This is assumed to be a fatal condition, and the called routine should not return. If the I/O error handler does return, the client process exits.

Note that the previous error handler is returned.

Input Device Functions

You can use the Xlib input device functions to:

- grab the pointer and individual buttons on the pointer
- grab the keyboard and individual keys on the keyboard
- move the pointer
- set the input focus
- manipulate the keyboard and pointer settings
- manipulate the keyboard encoding.

13.1   Pointer Grabbing

Xlib provides functions that you can use to control input from the pointer, which usually is a mouse. Usually, as soon as keyboard and mouse events occur, the X server delivers them to the appropriate client, which is determined by the window and input focus. The X server provides sufficient control over event delivery to allow window managers to support mouse ahead and various other styles of user interface. Many of these user interfaces depend upon synchronous delivery of events. The delivery of pointer and keyboard events can be controlled independently.

When mouse buttons or keyboard keys are grabbed, events will be sent to the grabbing client rather than the normal client who would have received the event. If the keyboard or pointer is in asynchronous mode, further mouse and keyboard events will continue to be processed. If the keyboard or pointer is in synchronous mode, no further events are processed until the grabbing client allows them (see *XAllowEvents*). The keyboard or pointer is considered frozen during this interval. The event that triggered the grab can also be replayed.

Note that the logical state of a device (as seen by client applications) may lag the physical state if device event processing is frozen.

There are two kinds of grabs: active and passive. An active grab occurs when a single client grabs the keyboard and/or pointer explicitly (see *XGrabPointer* and *XGrabKeyboard*). A passive grab occurs when clients grab a particular keyboard key or pointer button in a window, and the grab will activate when the key or button is actually pressed. Passive grabs are convenient for implementing reliable pop-up menus. For example, you can guarantee that the pop-up is mapped before the up pointer button event occurs by grabbing a button requesting synchronous behaviour. The down event will trigger the grab and freeze further processing of pointer events until you have the chance to map the pop-up window. You can then allow further event processing. The up event will then be correctly processed relative to the pop-up window.

For many operations, there are functions that take a time argument. The X server includes a timestamp in various events. One special time, called *CurrentTime*, represents the current server time. The X server maintains the time when the input focus was last changed, when the keyboard was last grabbed, when the pointer was last grabbed, or when a selection was last changed. Your application may be slow reacting to an event. You often need some way to specify that your request should not occur if another application has in the meanwhile taken control of the keyboard, pointer or selection. By providing the timestamp from the event in the request, you can arrange that the operation not take effect if someone else has performed an operation in the meanwhile.

A timestamp is a time value, expressed in milliseconds. It typically is the time since the last server reset. Timestamp values wrap around (after about 49.7 days). The server, given its current time is represented by timestamp T, always interprets timestamps from clients by treating half of the timestamp space as being later in time than T. One timestamp value, named *CurrentTime*, is never generated by the server. This value is reserved for use in requests to represent the current server time.

For many functions in this section, you pass pointer event mask bits. The valid pointer event mask bits are: *ButtonPressMask*, *ButtonReleaseMask*, *EnterWindowMask*, *LeaveWindowMask*, *PointerMotionMask*, *PointerMotionHintMask*, *Button1MotionMask*, *Button2MotionMask*, *Button3MotionMask*, *Button4MotionMask*, *Button5MotionMask*, *ButtonMotionMask* and *KeyMapStateMask*. For other functions in this section, you pass keymask bits. The valid keymask bits are: *ShiftMask*, *LockMask*, *ControlMask*, *Mod1Mask*, *Mod2Mask*, *Mod3Mask*, *Mod4Mask* and *Mod5Mask*.

To grab the pointer, use *XGrabPointer*.

```
int XGrabPointer(display, grab_window, owner_events,
        event_mask, pointer_mode, keyboard_mode,
        confine_to, cursor, time)
    Display *display;
    Window grab_window;
    Bool owner_events;
    unsigned int event_mask;
    int pointer_mode, keyboard_mode;
    Window confine_to;
    Cursor cursor;
    Time time;
```

display   Specifies the connection to the X server.

grab_window   Specifies the grab window.

owner_events   Specifies a Boolean value that indicates whether the pointer events are to be reported as usual or reported with respect to the grab window if selected by the event mask.

event_mask   Specifies which pointer events are reported to the client. The mask is the bitwise inclusive OR of the valid pointer event mask bits.

pointer_mode   Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync*.

keyboard_mode   Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync*.

confine_to   Specifies the window to confine the pointer in or *None*.

cursor   Specifies the cursor that is to be displayed during the grab or *None*.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XGrabPointer* function actively grabs control of the pointer and returns *GrabSuccess* if the grab was successful. Further pointer events are reported only to the grabbing client. *XGrabPointer* overrides any active pointer grab by this client. If owner_events is *False*, all generated pointer events are reported with respect to grab_window and are reported only if selected by event_mask. If owner_events is *True* and if a generated pointer event would normally be reported to this client, it is reported as usual. Otherwise, the event is reported with respect to the grab_window and is reported only if selected by event_mask. For either value of owner_events, unreported events are discarded.

If the pointer_mode is *GrabModeAsync*, pointer event processing continues as usual. If the pointer is currently frozen by this client, the processing of events for the pointer is resumed. If the pointer_mode is *GrabModeSync*, the state of the pointer, as seen by client applications, appears to freeze, and the X server generates no further pointer events until the grabbing client calls *XAllowEvents* or until the pointer grab is released. Actual pointer changes are not lost while the pointer is frozen; they are simply queued in the server for later processing.

If the keyboard_mode is *GrabModeAsync*, keyboard event processing is unaffected by activation of the grab. If the keyboard_mode is *GrabModeSync*, the state of the keyboard, as seen by client applications, appears to freeze, and the X server generates no further keyboard events until the grabbing client calls *XAllowEvents* or until the pointer grab is released. Actual keyboard changes are not lost while the pointer is frozen; they are simply queued in the server for later processing.

If a cursor is specified, it is displayed regardless of what window the pointer is in. If *None* is specified, the normal cursor for that window is displayed when the pointer is in grab_window or one of its subwindows; otherwise, the cursor for grab_window is displayed.

If a confine_to window is specified, the pointer is restricted to stay contained in that window. The confine_to window need have no relationship to the grab_window. If the pointer is not initially in the confine_to window, it is warped automatically to the closest edge just before the grab activates and enter/leave events are generated as usual. If the confine_to window is subsequently reconfigured, the pointer is warped automatically, as necessary, to keep it contained in the window.

The time argument allows you to avoid certain circumstances that come up if applications take a long time to respond or if there are long network delays. Consider a situation where you have two applications, both of which normally grab the pointer when clicked on. If both applications specify the timestamp from the event, the second application may wake up faster and successfully grab the pointer before the first application. The first application then will get an indication that the other application grabbed the pointer before its request was processed.

XGrabPointer generates *EnterNotify* and *LeaveNotify* events.

Either if grab_window or confine_to window is not viewable or if the confine_to window lies completely outside the boundaries of the root window, *XGrabPointer* fails and returns *GrabNotViewable*. If the pointer is actively grabbed by some other client, it fails and returns *AlreadyGrabbed*. If the pointer is frozen by an active grab of another client, it fails and returns *GrabFrozen*. If the specified time is earlier than the last-pointer-grab time or later than the current X server time, it fails and returns *GrabInvalidTime*. Otherwise, the last-pointer-grab time is set to the specified time (*CurrentTime* is replaced by the current X server time).

XGrabPointer can generate *BadCursor*, *BadValue* and *BadWindow* errors.

To ungrab the pointer, use *XUngrabPointer*.

```
XUngrabPointer(display, time)
      Display *display;
      Time time;
```

display   Specifies the connection to the X server.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XUngrabPointer* function releases the pointer and any queued events if this client has actively grabbed the pointer from *XGrabPointer*, *XGrabButton*, or from a normal button press. *XUngrabPointer* does not release the pointer if the specified time is earlier than the last-pointer-grab time or is later than the current X server time. It also generates *EnterNotify* and *LeaveNotify* events. The X server performs an *UngrabPointer* request automatically if the event window or confine_to window for an active pointer grab becomes not viewable or if window reconfiguration causes the confine_to window to lie completely outside the boundaries of the root window.

To change an active pointer grab, use *XChangeActivePointerGrab*.

```
XChangeActivePointerGrab(display, event_mask, cursor, time)
      Display *display;
      unsigned int event_mask;
      Cursor cursor;
      Time time;
```

display   Specifies the connection to the X server.

event_mask   Specifies which pointer events are reported to the client. The mask is the bitwise inclusive OR of the valid pointer event mask bits.

cursor   Specifies the cursor that is to be displayed or *None*.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XChangeActivePointerGrab* function changes the specified dynamic parameters if the pointer is actively grabbed by the client and if the specified time is no earlier than the last-pointer-grab time and no later than the current X server time. This function has no effect on the passive parameters of a *XGrabButton*. The interpretation of event_mask and cursor is the same as described in *XGrabPointer*.

XChangeActivePointerGrab can generate *BadCursor* and *BadValue* errors.

To grab a pointer button, use *XGrabButton*.

```
XGrabButton(display, button, modifiers, grab_window,
        owner_events, event_mask, pointer_mode,
        keyboard_mode, confine_to, cursor)
    Display *display;
    unsigned int button;
    unsigned int modifiers;
    Window grab_window;
    Bool owner_events;
    unsigned int event_mask;
    int pointer_mode, keyboard_mode;
    Window confine_to;
    Cursor cursor;
```

display   Specifies the connection to the X server.

button   Specifies the pointer button that is to be grabbed or *AnyButton*.

modifiers   Specifies the set of keymasks or *AnyModifier*. The mask is the bitwise inclusive OR of the valid keymask bits.

grab_window   Specifies the grab window.

owner_events   Specifies a Boolean value that indicates whether the pointer events are to be reported as usual or reported with respect to the grab window if selected by the event mask.

event_mask   Specifies which pointer events are reported to the client. The mask is the bitwise inclusive OR of the valid pointer event mask bits.

pointer_mode   Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync*.

keyboard_mode   Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync*.

confine_to   Specifies the window to confine the pointer in or *None*.

cursor   Specifies the cursor that is to be displayed or *None*.

The *XGrabButton* function establishes a passive grab. In the future, the pointer is actively grabbed (as for *XGrabPointer*), the last-pointer-grab time is set to the time at which the button was pressed (as transmitted in the *ButtonPress* event), and the *ButtonPress* event is reported if all of the following conditions are true:

- The pointer is not grabbed, and the specified button is logically pressed when the specified modifier keys are logically down, and no other buttons or modifier keys are logically down.

- The grab_window contains the pointer.

- The confine_to window (if any) is viewable.

- A passive grab on the same button/key combination does not exist on any ancestor of grab_window.

The interpretation of the remaining arguments is as for *XGrabPointer*. The active grab is terminated automatically when the logical state of the pointer has all buttons released (independent of the state of the logical modifier keys).

Note that the logical state of a device (as seen by client applications) may lag the physical state if device event processing is frozen.

This request overrides all previous grabs by the same client on the same button/key combinations on the same window. A modifiers of *AnyModifier* is equivalent to issuing the grab request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned KeyCodes. A button of *AnyButton* is equivalent to issuing the request for all possible buttons. Otherwise, it is not required that the specified button currently be assigned to a physical button.

If some other client has already issued a *XGrabButton* with the same button/key combination on the same window, a *BadAccess* error results. When using *AnyModifier* or *AnyButton*, the request fails completely, and a *BadAccess* error results (no grabs are established) if there is a conflicting grab for any combination. *XGrabButton* has no effect on an active grab.

XGrabButton can generate *BadCursor*, *BadValue* and *BadWindow* errors.

To ungrab a pointer button, use *XUngrabButton*.

```
XUngrabButton(display, button, modifiers, grab_window)
      Display *display;
      unsigned int button;
      unsigned int modifiers;
      Window grab_window;
```

display   Specifies the connection to the X server.

button   Specifies the pointer button that is to be released or *AnyButton*.

modifiers   Specifies the set of keymasks or *AnyModifier*. The mask is the bitwise inclusive OR of the valid keymask bits.

grab_window   Specifies the grab window.

The *XUngrabButton* function releases the passive button/key combination on the specified window if it was grabbed by this client. A modifiers of *AnyModifier* is equivalent to issuing the ungrab request for all possible modifier combinations, including the combination of no modifiers. A button of *AnyButton* is equivalent to issuing the request for all possible buttons. *XUngrabButton* has no effect on an active grab.

XUngrabButton can generate *BadValue* and *BadWindow* errors.

13.2   Keyboard Grabbing

Xlib provides functions that you can use to grab or ungrab the keyboard as well as allow events.

For many functions in this section, you pass keymask bits. The valid keymask bits are: *ShiftMask*, *LockMask*, *ControlMask*, *Mod1Mask*, *Mod2Mask*, *Mod3Mask*, *Mod4Mask* and *Mod5Mask*.

To grab the keyboard, use *XGrabKeyboard*.

```
int XGrabKeyboard(display, grab_window, owner_events,
        pointer_mode, keyboard_mode, time)
    Display *display;
    Window grab_window;
    Bool owner_events;
    int pointer_mode, keyboard_mode;
    Time time;
```

display   Specifies the connection to the X server.

grab_window   Specifies the grab window.

owner_events   Specifies a Boolean value that indicates whether the keyboard events are to be reported as usual.

pointer_mode   Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync*.

keyboard_mode   Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync*.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XGrabKeyboard* function actively grabs control of the keyboard and generates *FocusIn* and *FocusOut* events. Further key events are reported only to the grabbing client. *XGrabKeyboard* overrides any active keyboard grab by this client. If owner_events is *False*, all generated key events are reported with respect to grab_window. If owner_events is *True* and if a generated key event would normally be reported to this client, it is reported normally; otherwise, the event is reported with respect to the grab_window. Both *KeyPress* and *KeyRelease* events are always reported, independent of any event selection made by the client.

If the keyboard_mode argument is *GrabModeAsync*, keyboard event processing continues as usual. If the keyboard is currently frozen by this client, then processing of keyboard events is resumed. If the keyboard_mode argument is *GrabModeSync*, the state of the keyboard (as seen by client applications) appears to freeze, and the X server generates no further keyboard events until the grabbing client issues a releasing *XAllowEvents* call or until the keyboard grab is released. Actual keyboard changes are not lost while the keyboard is frozen; they are simply queued in the server for later processing.

If pointer_mode is *GrabModeAsync*, pointer event processing is unaffected by activation of the grab. If pointer_mode is *GrabModeSync*, the state of the pointer (as seen by client applications) appears to freeze, and the X server generates no further pointer events until the grabbing client issues a releasing *XAllowEvents* call or until the keyboard grab is released. Actual pointer changes are not lost while the pointer is frozen; they are simply queued in the server for later processing.

If the keyboard is actively grabbed by some other client, *XGrabKeyboard* fails and returns *AlreadyGrabbed*. If grab_window is not viewable, it fails and returns *GrabNotViewable*. If the keyboard is frozen by an active grab of another client, it fails and returns *GrabFrozen*. If the specified time is earlier than the last-keyboard-grab time or later than the current X server time,

it fails and returns *GrabInvalidTime*. Otherwise, the last-keyboard-grab time is set to the specified time (*CurrentTime* is replaced by the current X server time).

XGrabKeyboard can generate *BadValue* and *BadWindow* errors.

To ungrab the keyboard, use *XUngrabKeyboard*.

```
XUngrabKeyboard(display, time)
      Display *display;
      Time time;
```

display   Specifies the connection to the X server.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XUngrabKeyboard* function releases the keyboard and any queued events if this client has it actively grabbed from either *XGrabKeyboard* or *XGrabKey*. *XUngrabKeyboard* does not release the keyboard and any queued events if the specified time is earlier than the last-keyboard-grab time or is later than the current X server time. It also generates *FocusIn* and *FocusOut* events. The X server automatically performs an *UngrabKeyboard* request if the event window for an active keyboard grab becomes not viewable.

To passively grab a single key of the keyboard, use *XGrabKey*.

```
XGrabKey(display, keycode, modifiers, grab_window,
          owner_events, pointer_mode, keyboard_mode)
      Display *display;
      int keycode;
      unsigned int modifiers;
      Window grab_window;
      Bool owner_events;
      int pointer_mode, keyboard_mode;
```

display   Specifies the connection to the X server.

keycode   Specifies the KeyCode or *AnyKey*.

modifiers   Specifies the set of keymasks or *AnyModifier*. The mask is the bitwise inclusive OR of the valid keymask bits.

grab_window   Specifies the grab window.

owner_events   Specifies a Boolean value that indicates whether the keyboard events are to be reported as usual.

pointer_mode   Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync*.

keyboard_mode   Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync*.

The *XGrabKey* function establishes a passive grab on the keyboard. In the future, the keyboard is actively grabbed (as for *XGrabKeyboard*), the last-keyboard-grab time is set to the time at which the key was pressed (as transmitted in the *KeyPress* event), and the *KeyPress* event is reported if all of the following conditions are true:

- The keyboard is not grabbed and the specified key (which can itself be a modifier key) is logically pressed when the specified modifier keys are logically down, and no other modifier keys are logically down.

- Either the grab_window is an ancestor of (or is) the focus window, or the grab_window is a descendant of the focus window and contains the pointer.

- A passive grab on the same key combination does not exist on any ancestor of grab_window.

The interpretation of the remaining arguments is as for *XGrabKeyboard*. The active grab is terminated automatically when the logical state of the keyboard has the specified key released (independent of the logical state of the modifier keys).

Note that the logical state of a device (as seen by client applications) may lag the physical state if device event processing is frozen.

A modifiers argument of *AnyModifier* is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned KeyCodes. A keycode argument of *AnyKey* is equivalent to issuing the request for all possible KeyCodes. Otherwise, the specified keycode must be in the range specified by min_keycode and max_keycode in the connection setup, or a *BadValue* error results.

If some other client has issued a *XGrabKey* with the same key combination on the same window, a *BadAccess* error results. When using *AnyModifier* or *AnyKey*, the request fails completely, and a *BadAccess* error results (no grabs are established) if there is a conflicting grab for any combination.

XGrabKey can generate *BadAccess*, *BadValue* and *BadWindow* errors.

To ungrab a key, use *XUngrabKey*.

```
XUngrabKey(display, keycode, modifiers, grab_window)
      Display *display;
      int keycode;
      unsigned int modifiers;
      Window grab_window;
```

display   Specifies the connection to the X server.

keycode   Specifies the KeyCode or *AnyKey*.

modifiers   Specifies the set of keymasks or *AnyModifier*. The mask is the bitwise inclusive OR of the valid keymask bits.

grab_window   Specifies the grab window.

The *XUngrabKey* function releases the key combination on the specified window if it was grabbed by this client. It has no effect on an active grab. A modifiers of *AnyModifier* is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). A keycode argument of *AnyKey* is equivalent to issuing the request for all possible key codes.

XUngrabKey can generate *BadValue* and *BadWindow* errors.

13.3   Resuming Event Processing

The previous sections discussed grab mechanisms with which processing of events by the server can be temporarily suspended. This section describes the mechanism for resuming event processing.

To allow further events to be processed when the device has been frozen, use *XAllowEvents*.

```
XAllowEvents(display, event_mode, time)
       Display *display;
       int event_mode;
       Time time;
```

display   Specifies the connection to the X server.

event_mode   Specifies the event mode. You can pass *AsyncPointer*, *SyncPointer*, *AsyncKeyboard*, *SyncKeyboard*, *ReplayPointer*, *ReplayKeyboard*, *AsyncBoth* or *SyncBoth*.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XAllowEvents* function releases some queued events if the client has caused a device to freeze. It has no effect if the specified time is earlier than the last-grab time of the most recent active grab for the client or if the specified time is later than the current X server time. Depending on the event_mode argument, the following occurs:

AsyncPointer   If the pointer is frozen by the client, pointer event processing continues as usual. If the pointer is frozen twice by the client on behalf of two separate grabs, *AsyncPointer* thaws for both. *AsyncPointer* has no effect if the pointer is not frozen by the client, but the pointer need not be grabbed by the client.

SyncPointer   If the pointer is frozen and actively grabbed by the client, pointer event processing continues as usual until the next *ButtonPress* or *ButtonRelease* event is reported to the client. At this time, the pointer again appears to freeze. However, if the reported event causes the pointer grab to be released, the pointer does not freeze. *SyncPointer* has no effect if the pointer is not frozen by the client or if the pointer is not grabbed by the client.

ReplayPointer   If the pointer is actively grabbed by the client and is frozen as the result of an event having been sent to the client (either from the activation of a *XGrabButton* or from a previous *XAllowEvents* with mode *SyncPointer* but not from a *XGrabPointer*), the pointer grab is released and that event is completely reprocessed. This time, however, the function ignores any passive grabs at or above (towards the root of) the grab_window of the grab just released. The request has no effect if the pointer is not grabbed by the client or if the pointer is not frozen as the result of an event.

AsyncKeyboard   If the keyboard is frozen by the client, keyboard event processing continues as usual. If the keyboard is frozen twice by the client on behalf of two separate grabs, *AsyncKeyboard* thaws for both. *AsyncKeyboard* has no effect if the keyboard is not frozen by the client, but the keyboard need not be grabbed by the client.

SyncKeyboard   If the keyboard is frozen and actively grabbed by the client, keyboard event processing continues as usual until the next *KeyPress* or *KeyRelease* event is reported to the client. At this time, the keyboard again appears to freeze. However, if the reported event causes the keyboard grab to be released, the keyboard does not freeze. *SyncKeyboard* has no effect if the keyboard is not

frozen by the client or if the keyboard is not grabbed by the client.

ReplayKeyboard   If the keyboard is actively grabbed by the client and is frozen as the result of an event having been sent to the client (either from the activation of a *XGrabKey* or from a previous *XAllowEvents* with mode *SyncKeyboard* but not from a *XGrabKeyboard*), the keyboard grab is released and that event is completely reprocessed. This time, however, the function ignores any passive grabs at or above (towards the root of) the grab_window of the grab just released. The request has no effect if the keyboard is not grabbed by the client or if the keyboard is not frozen as the result of an event.

SyncBoth   If both pointer and keyboard are frozen by the client, event processing for both devices continues as usual until the next *ButtonPress*, *ButtonRelease*, *KeyPress* or *KeyRelease* event is reported to the client for a grabbed device (button event for the pointer, key event for the keyboard), at which time the devices again appear to freeze. However, if the reported event causes the grab to be released, then the devices do not freeze (but if the other device is still grabbed, then a subsequent event for it will still cause both devices to freeze). *SyncBoth* has no effect unless both pointer and keyboard are frozen by the client. If the pointer or keyboard is frozen twice by the client on behalf of two separate grabs, *SyncBoth* thaws for both (but a subsequent freeze for *SyncBoth* will only freeze each device once).

AsyncBoth   If the pointer and the keyboard are frozen by the client, event processing for both devices continues as usual. If a device is frozen twice by the client on behalf of two separate grabs, *AsyncBoth* thaws for both. *AsyncBoth* has no effect unless both pointer and keyboard are frozen by the client.

AsyncPointer, *SyncPointer* and *ReplayPointer* have no effect on the processing of keyboard events. *AsyncKeyboard*, *SyncKeyboard* and *ReplayKeyboard* have no effect on the processing of pointer events. It is possible for both a pointer grab and a keyboard grab (by the same or different clients) to be active simultaneously. If a device is frozen on behalf of either grab, no event processing is performed for the device. It is possible for a single device to be frozen because of both grabs. In this case, the freeze must be released on behalf of both grabs before events can again be processed. If a device is frozen twice by a single client, then a single *AllowEvents* releases both.

XAllowEvents can generate a *BadValue* error.

13.4   Moving the Pointer

Although movement of the pointer normally should be left to the control of the end user, sometimes it is necessary to move the pointer to a new position under program control.

To move the pointer to an arbitrary point in a window, use *XWarpPointer*.

```
XWarpPointer(display, src_w, dest_w, src_x, src_y,
            src_width, src_height, dest_x, dest_y)
        Display *display;
        Window src_w, dest_w;
        int src_x, src_y;
        unsigned int src_width, src_height;
        int dest_x, dest_y;
```

display   Specifies the connection to the X server.

src_w   Specifies the source window or *None*.

dest_w   Specifies the destination window or *None*.

src_x
src_y
src_width
src_height   Specify a rectangle in the source window.

dest_x
dest_y   Specify the x and y coordinates within the destination window.

If dest_w is *None*, *XWarpPointer* moves the pointer by the offsets (dest_x, dest_y) relative to the current position of the pointer. If dest_w is a window, *XWarpPointer* moves the pointer to the offsets (dest_x, dest_y) relative to the origin of dest_w. However, if src_w is a window, the move only takes place if the window src_w contains the pointer and if the specified rectangle of src_w contains the pointer.

The src_x and src_y coordinates are relative to the origin of src_w. If src_height is zero, it is replaced with the current height of src_w minus src_y. If src_width is zero, it is replaced with the current width of src_w minus src_x.

There is seldom any reason for calling this function. The pointer should normally be left to the user. If you do use this function, however, it generates events just as if the user had instantaneously moved the pointer from one position to another. Note that you cannot use *XWarpPointer* to move the pointer outside the confine_to window of an active pointer grab. An attempt to do so will only move the pointer as far as the closest edge of the confine_to window.

XWarpPointer can generate a *BadWindow* error.

13.5   Controlling Input Focus

Xlib provides functions that you can use to set and get the input focus. The input focus is a shared resource, and cooperation among clients is required for correct interaction. See the **ICCCM** specification for input focus policy.

To set the input focus, use *XSetInputFocus*.

```
XSetInputFocus(display, focus, revert_to, time)
      Display *display;
      Window focus;
      int revert_to;
      Time time;
```

display   Specifies the connection to the X server.

focus   Specifies the window, *PointerRoot*, or *None*.

revert_to   Specifies where the input focus reverts to if the window becomes not viewable. You can pass *RevertToParent*, *RevertToPointerRoot* or *RevertToNone*.

time   Specifies the time. You can pass either a timestamp or *CurrentTime*.

The *XSetInputFocus* function changes the input focus and the last-focus-change time. It has no effect if the specified time is earlier than the current last-focus-change time or is later than the current X server time. Otherwise, the last-focus-change time is set to the specified time (*CurrentTime* is replaced by the current X server time). *XSetInputFocus* causes the X server to generate *FocusIn* and *FocusOut* events.

Depending on the focus argument, the following occurs:

- If focus is *None*, all keyboard events are discarded until a new focus window is set, and the revert_to argument is ignored.

- If focus is a window, it becomes the keyboard's focus window. If a generated keyboard event would normally be reported to this window or one of its inferiors, the event is reported as usual. Otherwise, the event is reported relative to the focus window.

- If focus is *PointerRoot*, the focus window is dynamically taken to be the root window of whatever screen the pointer is on at each keyboard event. In this case, the revert_to argument is ignored.

The specified focus window must be viewable at the time *XSetInputFocus* is called, or a *BadMatch* error results. If the focus window later becomes not viewable, the X server evaluates the revert_to argument to determine the new focus window as follows:

- If revert_to is *RevertToParent*, the focus reverts to the parent (or the closest viewable ancestor), and the new revert_to value is taken to be *RevertToNone*.

- If revert_to is *RevertToPointerRoot* or *RevertToNone*, the focus reverts to *PointerRoot* or *None*, respectively. When the focus reverts, the X server generates *FocusIn* and *FocusOut* events, but the last-focus-change time is not affected.

XSetInputFocus can generate *BadMatch*, *BadValue* and *BadWindow* errors.

To obtain the current input focus, use *XGetInputFocus*.

```
XGetInputFocus(display, focus_return, revert_to_return)
      Display *display;
      Window *focus_return;
      int *revert_to_return;
```

display   Specifies the connection to the X server.

focus_return   Returns the focus window, *PointerRoot*, or *None*.

revert_to_return   Returns the current focus state (*RevertToParent*, *RevertToPointerRoot* or *RevertToNone*).

The *XGetInputFocus* function returns the focus window and the current focus state.

13.6   Keyboard and Pointer Settings

Xlib provides functions that you can use to change the keyboard control, obtain a list of the auto-repeat keys, turn keyboard auto-repeat on or off, ring the bell, set or obtain the pointer button or keyboard mapping, and obtain a bit vector for the keyboard.

This section discusses the user-preference options of bell, key click, pointer behaviour, and so on. The default values for many of these functions are determined by command line arguments to the X server and, on POSIX-conformant systems, are typically set in the */etc/ttys* file. Not all implementations will actually be able to control all of these parameters.

The *XChangeKeyboardControl* function changes control of a keyboard and operates on a *XKeyboardControl* structure:

```
/* Mask bits for ChangeKeyboardControl */
      #define KBKeyClickPercent   (1L<<0)
      #define KBBellPercent       (1L<<1)
      #define KBBellPitch         (1L<<2)
      #define KBBellDuration      (1L<<3)
      #define KBLed               (1L<<4)
      #define KBLedMode           (1L<<5)
      #define KBKey               (1L<<6)
      #define KBAutoRepeatMode    (1L<<7)

/* Values */


typedef struct {
      int key_click_percent;
      int bell_percent;
      int bell_pitch;
      int bell_duration;
      int led;
      int led_mode;              /* LedModeOn, LedModeOff */
      int key;
      int auto_repeat_mode;  /* AutoRepeatModeOff, AutoRepeatModeOn,
                                AutoRepeatModeDefault */
} XKeyboardControl;
```

The key_click_percent member sets the volume for key clicks between 0 (off) and 100 (loud) inclusive, if possible. A setting of −1 restores the default. Other negative values generate a *BadValue* error.

The bell_percent sets the base volume for the bell between 0 (off) and 100 (loud) inclusive, if possible. A setting of −1 restores the default. Other negative values generate a *BadValue* error. The bell_pitch member sets the pitch (specified in Hz) of the bell, if possible. A setting of −1 restores the default. Other negative values generate a *BadValue* error. The bell_duration member sets the duration of the bell specified in milliseconds, if possible. A setting of −1 restores the default. Other negative values generate a *BadValue* error.

If both the led_mode and led members are specified, the state of that LED is changed, if possible. The led_mode member can be set to *LedModeOn* or *LedModeOff*. If only led_mode is specified, the state of all LEDs are changed, if possible. At most 32 LEDs numbered from one are supported. No standard interpretation of LEDs is defined. If led is specified without led_mode, a *BadMatch* error results.

If both the auto_repeat_mode and key members are specified, the auto_repeat_mode of that key is changed (according to *AutoRepeatModeOn*, *AutoRepeatModeOff* or *AutoRepeatModeDefault*), if

possible. If only auto_repeat_mode is specified, the global auto_repeat_mode for the entire keyboard is changed, if possible, and does not affect the per key settings. If a key is specified without an auto_repeat_mode, a *BadMatch* error results. Each key has an individual mode of whether or not it should auto-repeat and a default setting for the mode. In addition, there is a global mode of whether auto-repeat should be enabled or not and a default setting for that mode. When global mode is *AutoRepeatModeOn*, keys should obey their individual auto-repeat modes. When global mode is *AutoRepeatModeOff*, no keys should auto-repeat. An auto-repeating key generates alternating *KeyPress* and *KeyRelease* events. When a key is used as a modifier, it is desirable for the key not to auto-repeat, regardless of its auto-repeat setting.

A bell generator connected with the console but not directly on a keyboard is treated as if it were part of the keyboard. The order in which controls are verified and altered is server-dependent. If an error is generated, a subset of the controls may have been altered.

```
XChangeKeyboardControl(display, value_mask, values)
      Display *display;
      unsigned long value_mask;
      XKeyboardControl *values;
```

display   Specifies the connection to the X server.

value_mask   Specifies which controls to change. This mask is the bitwise inclusive OR of the valid control mask bits.

values   Specifies one value for each bit set to 1 in the mask.

The *XChangeKeyboardControl* function controls the keyboard characteristics defined by the *XKeyboardControl* structure. The value_mask argument specifies which values are to be changed.

XChangeKeyboardControl can generate *BadMatch* and *BadValue* errors.

To obtain the current control values for the keyboard, use *XGetKeyboardControl*.

```
XGetKeyboardControl(display, values_return)
      Display *display;
      XKeyboardState *values_return;
```

display   Specifies the connection to the X server.

values_return   Returns the current keyboard controls in the specified *XKeyboardState* structure.

The *XGetKeyboardControl* function returns the current control values for the keyboard to the *XKeyboardState* structure.

```
typedef struct {
      int key_click_percent;
      int bell_percent;
      unsigned int bell_pitch, bell_duration;
      unsigned long led_mask;
      int global_auto_repeat;
      char auto_repeats[32];
} XKeyboardState;
```

For the LEDs, the least-significant bit of led_mask corresponds to LED one, and each bit set to 1 in led_mask indicates an LED that is lit. The global_auto_repeat member can be set to *AutoRepeatModeOn* or *AutoRepeatModeOff*. The auto_repeats member is a bit vector. Each bit set to 1 indicates that auto-repeat is enabled for the corresponding key. The vector is represented as

32 bytes. Byte N (from 0) contains the bits for keys 8N to 8N + 7 with the least-significant bit in the byte representing key 8N.

To turn on keyboard auto-repeat, use *XAutoRepeatOn*.

```
XAutoRepeatOn(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XAutoRepeatOn* function turns on auto-repeat for the keyboard on the specified display.

To turn off keyboard auto-repeat, use *XAutoRepeatOff*.

```
XAutoRepeatOff(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XAutoRepeatOff* function turns off auto-repeat for the keyboard on the specified display.

To ring the bell, use *XBell*.

```
XBell(display, percent)
      Display *display;
      int percent;
```

display   Specifies the connection to the X server.

percent   Specifies the volume for the bell, which can range from −100 to 100 inclusive.

The *XBell* function rings the bell on the keyboard on the specified display, if possible. The specified volume is relative to the base volume for the keyboard. If the value for the percent argument is not in the range −100 to 100 inclusive, a *BadValue* error results. The volume at which the bell rings when the percent argument is non-negative is:

```
base − [(base * percent) / 100] + percent
```

The volume at which the bell rings when the percent argument is negative is:

```
base + [(base * percent) / 100]
```

To change the base volume of the bell, use *XChangeKeyboardControl*.

XBell can generate a *BadValue* error.

To obtain a bit vector that describes the state of the keyboard, use *XQueryKeymap*.

```
XQueryKeymap(display, keys_return)
      Display *display;
      char keys_return[32];
```

display   Specifies the connection to the X server.

keys_return   Returns an array of bytes that identifies which keys are pressed down. Each bit represents one key of the keyboard.

The *XQueryKeymap* function returns a bit vector for the logical state of the keyboard, where each bit set to 1 indicates that the corresponding key is currently pressed down. The vector is represented as 32 bytes. Byte N (from 0) contains the bits for keys 8N to 8N + 7 with the least-significant bit in the byte representing key 8N.

Note that the logical state of a device (as seen by client applications) may lag the physical state if device event processing is frozen.

To set the mapping of the pointer buttons, use *XSetPointerMapping*.

```
int XSetPointerMapping(display, map, nmap)
      Display *display;
      unsigned char map[];
      int nmap;
```

display   Specifies the connection to the X server.

map   Specifies the mapping list.

nmap   Specifies the number of items in the mapping list.

The *XSetPointerMapping* function sets the mapping of the pointer. If it succeeds, the X server generates a *MappingNotify* event, and *XSetPointerMapping* returns *MappingSuccess*. Element map[i] defines the logical button number for the physical button i+1. The length of the list must be the same as *XGetPointerMapping* would return, or a *BadValue* error results. A zero element disables a button, and elements are not restricted in value by the number of physical buttons. However, no two elements can have the same non-zero value, or a *BadValue* error results. If any of the buttons to be altered are logically in the down state, *XSetPointerMapping* returns *MappingBusy*, and the mapping is not changed.

XSetPointerMapping can generate a *BadValue* error.

To get the pointer mapping, use *XGetPointerMapping*.

```
int XGetPointerMapping(display, map_return, nmap)
      Display *display;
      unsigned char map_return[];
      int nmap;
```

display   Specifies the connection to the X server.

map_return   Returns the mapping list.

nmap   Specifies the number of items in the mapping list.

The *XGetPointerMapping* function returns the current mapping of the pointer. Pointer buttons are numbered starting from one. *XGetPointerMapping* returns the number of physical buttons actually on the pointer. The nominal mapping for a pointer is map[i]=i+1. The nmap argument specifies the length of the array where the pointer mapping is returned, and only the first nmap elements are returned in map_return.

To control the pointer's interactive feel, use *XChangePointerControl*.

```
XChangePointerControl(display, do_accel, do_threshold,
        accel_numerator, accel_denominator, threshold)
      Display *display;
      Bool do_accel, do_threshold;
      int accel_numerator, accel_denominator;
      int threshold;
```

display   Specifies the connection to the X server.

do_accel   Specifies a Boolean value that controls whether the values for the accel_numerator or accel_denominator are used.

do_threshold   Specifies a Boolean value that controls whether the value for the threshold is used.

accel_numerator Specifies the numerator for the acceleration multiplier.

accel_denominator Specifies the denominator for the acceleration multiplier.

threshold Specifies the acceleration threshold.

The *XChangePointerControl* function defines how the pointing device moves. The acceleration, expressed as a fraction, is a multiplier for movement. For example, specifying 3/1 means the pointer moves three times as fast as normal. The fraction may be rounded arbitrarily by the X server. Acceleration only takes effect if the pointer moves more than threshold pixels at once and only applies to the amount beyond the value in the threshold argument. Setting a value to −1 restores the default. The values of the do_accel and do_threshold arguments must be *True* for the pointer values to be set, or the parameters are unchanged. Negative values (other than −1) generate a *BadValue* error, as does a zero value for the accel_denominator argument.

XChangePointerControl can generate a *BadValue* error.

To get the current pointer parameters, use *XGetPointerControl*.

```
XGetPointerControl(display, accel_numerator_return,
        accel_denominator_return, threshold_return)
    Display *display;
    int *accel_numerator_return, *accel_denominator_return;
    int *threshold_return;
```

display Specifies the connection to the X server.

accel_numerator_return
        Returns the numerator for the acceleration multiplier.

accel_denominator_return
        Returns the denominator for the acceleration multiplier.

threshold_return Returns the acceleration threshold.

The *XGetPointerControl* function returns the pointer's current acceleration multiplier and acceleration threshold.

13.7   Keyboard Encoding

A KeyCode represents a physical (or logical) key. KeyCodes lie in the inclusive range [8,255]. A KeyCode value carries no intrinsic information, although server implementors may attempt to encode geometry (for example, matrix) information in some fashion so that it can be interpreted in a server-dependent fashion. The mapping between keys and KeyCodes cannot be changed.

A KeySym is an encoding of a symbol on the cap of a key. The set of defined KeySyms includes the ISO Latin character sets (1–4), Katakana, Arabic, Cyrillic, Greek, Technical, Special, Publishing, APL, Hebrew, and a special miscellany of keys found on keyboards (Return, Help, Tab, and so on). To the extent possible, these sets are derived from international standards. In areas where no standards exist, some of these sets are derived from Digital Equipment Corporation standards. The list of defined symbols can be found in **<X11/keysymdef.h>**. Unfortunately, some C preprocessors have limits on the number of defined symbols. If you must use KeySyms not in the Latin 1–4, Greek, and miscellaneous classes, you may have to define a symbol for those sets. Most applications usually only include **<X11/keysym.h>**, which defines symbols for ISO Latin 1–4, Greek and miscellaneous.

A list of KeySyms is associated with each KeyCode. The list is intended to convey the set of symbols on the corresponding key. If the list (ignoring trailing NoSymbol entries) is a single KeySym ''*K*'', then the list is treated as if it were the list ''*K* NoSymbol *K* NoSymbol''. If the list (ignoring trailing NoSymbol entries) is a pair of KeySyms ''*K1 K2*'', then the list is treated as if it were the list ''*K1 K2 K1 K2*''. If the list (ignoring trailing NoSymbol entries) is a triple of KeySyms ''*K1 K2 K3*'', then the list is treated as if it were the list ''*K1 K2 K3* NoSymbol''. When an explicit ''void'' element is desired in the list, the value VoidSymbol can be used.

The first four elements of the list are split into two groups of KeySyms. Group 1 contains the first and second KeySyms; Group 2 contains the third and fourth KeySyms. Within each group, if the second element of the group is NoSymbol, then the group should be treated as if the second element were the same as the first element, except when the first element is an alphabetic KeySym ''*K*'' for which both lower-case and upper-case forms are defined. In that case, the group should be treated as if the first element were the lower-case form of ''*K*'' and the second element were the upper-case form of ''*K*''.

The standard rules for obtaining a KeySym from a *KeyPress* event make use of only the Group 1 and Group 2 KeySyms; no interpretation of other KeySyms in the list is given. Which group to use is determined by the modifier state. Switching between groups is controlled by the KeySym named MODE SWITCH, by attaching that KeySym to some KeyCode and attaching that KeyCode to any one of the modifiers Mod1 through Mod5. This modifier is called the ''group modifier''. For any KeyCode, Group 1 is used when the group modifier is off, and Group 2 is used when the group modifier is on.

Within a group, the modifier state also determines which KeySym to use. The first KeySym is used when the Shift and Lock modifiers are off. The second KeySym is used when the Shift modifier is on, when the Lock modifier is on and the second KeySym is upper-case alphabetic, or when the Lock modifier is on and is interpreted as ShiftLock. Otherwise, when the Lock modifier is on and is interpreted as CapsLock, the state of the Shift modifier is applied first to select a KeySym; but if that KeySym is lower-case alphabetic, then the corresponding upper-case KeySym is used instead.

No spatial geometry of the symbols on the key is defined by their order in the KeySym list, although a geometry might be defined on a vendor-specific basis. The X server does not use the mapping between KeyCodes and KeySyms. Rather, it stores it merely for reading and writing by clients.

The KeyMask modifier named Lock is intended to be mapped to either a CapsLock or a ShiftLock key, but which one is left as application-specific and/or user-specific. However, it is suggested that the determination be made according to the associated KeySym(s) of the corresponding KeyCode.

To obtain the legal KeyCodes for a display, use *XDisplayKeycodes*.

```
XDisplayKeycodes(display, min_keycodes_return, max_keycodes_return)
        Display *display;
        int *min_keycodes_return, *max_keycodes_return;
```

display   Specifies the connection to the X server.

min_keycodes_return
                Returns the minimum number of KeyCodes.

max_keycodes_return
                Returns the maximum number of KeyCodes.

The *XDisplayKeycodes* function returns the min-keycodes and max-keycodes supported by the specified display. The minimum number of KeyCodes returned is never less than 8, and the maximum number of KeyCodes returned is never greater than 255. Not all KeyCodes in this range are required to have corresponding keys.

To obtain the symbols for the specified KeyCodes, use *XGetKeyboardMapping*.

```
KeySym *XGetKeyboardMapping(display, first_keycode,
        keycode_count, keysyms_per_keycode_return)
    Display *display;
    KeyCode first_keycode;
    int keycode_count;
    int *keysyms_per_keycode_return;
```

display   Specifies the connection to the X server.

first_keycode   Specifies the first KeyCode that is to be returned.

keycode_count   Specifies the number of KeyCodes that are to be returned.

keysyms_per_keycode_return
                Returns the number of KeySyms per KeyCode.

The *XGetKeyboardMapping* function returns the symbols for the specified number of KeyCodes starting with first_keycode. The value specified in first_keycode must be greater than or equal to min_keycode as returned by *XDisplayKeycodes*, or a *BadValue* error results. In addition, the following expression must be less than or equal to max_keycode as returned by *XDisplayKeycodes*:

```
first_keycode + keycode_count − 1
```

If this is not the case, a *BadValue* error results. The number of elements in the KeySyms list is:

```
keycode_count * keysyms_per_keycode_return
```

KeySym number N, counting from zero, for KeyCode K has the following index in the list, counting from zero:

```
(K − first_code) * keysyms_per_code_return + N
```

The X server arbitrarily chooses the keysyms_per_keycode_return value to be large enough to report all requested symbols. A special KeySym value of *NoSymbol* is used to fill in unused elements for individual KeyCodes. To free the storage returned by *XGetKeyboardMapping*, use

XFree.

XGetKeyboardMapping can generate a *BadValue* error.

To change the keyboard mapping, use *XChangeKeyboardMapping*.

```
XChangeKeyboardMapping(display, first_keycode, keysyms_per_keycode,
        keysyms, num_codes)
    Display *display;
    int first_keycode;
    int keysyms_per_keycode;
    KeySym *keysyms;
    int num_codes;
```

display   Specifies the connection to the X server.

first_keycode   Specifies the first KeyCode that is to be changed.

keysyms_per_keycode
                     Specifies the number of KeySyms per KeyCode.

keysyms   Specifies an array of KeySyms.

num_codes   Specifies the number of KeyCodes that are to be changed.

The *XChangeKeyboardMapping* function defines the symbols for the specified number of KeyCodes starting with first_keycode. The symbols for KeyCodes outside this range remain unchanged. The number of elements in keysyms must be:

```
num_codes * keysyms_per_keycode
```

The specified first_keycode must be greater than or equal to min_keycode returned by *XDisplayKeycodes*, or a *BadValue* error results. In addition, the following expression must be less than or equal to max_keycode as returned by *XDisplayKeycodes*, or a *BadValue* error results:

```
first_keycode + num_codes − 1
```

KeySym number N, counting from zero, for KeyCode K has the following index in keysyms, counting from zero:

```
(K − first_keycode) * keysyms_per_keycode + N
```

The specified keysyms_per_keycode can be chosen arbitrarily by the client to be large enough to hold all desired symbols. A special KeySym value of *NoSymbol* should be used to fill in unused elements for individual KeyCodes. It is legal for *NoSymbol* to appear in nontrailing positions of the effective list for a KeyCode. *XChangeKeyboardMapping* generates a *MappingNotify* event.

There is no requirement that the X server interpret this mapping. It is merely stored for reading and writing by clients.

XChangeKeyboardMapping can generate *BadAlloc* and *BadValue* errors.

The next four functions make use of the *XModifierKeymap* data structure, which contains:

```
typedef struct {
    int max_keypermod;    /* This server's max number of keys
                             per modifier */
    KeyCode *modifiermap; /* An 8 by max_keypermod array of the
                             modifiers */
} XModifierKeymap;
```

To create an *XModifierKeymap* structure, use *XNewModifiermap*.

```
XModifierKeymap *XNewModifiermap(max_keys_per_mod)
        int max_keys_per_mod;
```

max_keys_per_mod
                Specifies the number of KeyCode entries preallocated to the modifiers in the
                map.

The *XNewModifiermap* function returns a pointer to *XModifierKeymap* structure for later use.

To add a new entry to an *XModifierKeymap* structure, use *XInsertModifiermapEntry*.

```
XModifierKeymap *XInsertModifiermapEntry(modmap, keycode_entry,
        modifier)
    XModifierKeymap *modmap;
    KeyCode keycode_entry;
    int modifier;
```

modmap   Specifies the *XModifierKeymap* structure.

keycode_entry   Specifies the KeyCode.

modifier   Specifies the modifier.

The *XInsertModifiermapEntry* function adds the specified KeyCode to the set that controls the
specified modifier and returns the resulting *XModifierKeymap* structure (expanded as needed).

To delete an entry from an *XModifierKeymap* structure, use *XDeleteModifiermapEntry*.

```
XModifierKeymap *XDeleteModifiermapEntry(modmap, keycode_entry,
        modifier)
    XModifierKeymap *modmap;
    KeyCode keycode_entry;
    int modifier;
```

modmap   Specifies the *XModifierKeymap* structure.

keycode_entry   Specifies the KeyCode.

modifier   Specifies the modifier.

The *XDeleteModifiermapEntry* function deletes the specified KeyCode from the set that controls
the specified modifier and returns a pointer to the resulting *XModifierKeymap* structure.

To destroy an *XModifierKeymap* structure, use *XFreeModifiermap*.

```
XFreeModifiermap(modmap)
        XModifierKeymap *modmap;
```

modmap   Specifies the *XModifierKeymap* structure.

The *XFreeModifiermap* function frees the specified *XModifierKeymap* structure.

To set the KeyCodes to be used as modifiers, use *XSetModifierMapping*.

```
int XSetModifierMapping(display, modmap)
        Display *display;
        XModifierKeymap *modmap;
```

display   Specifies the connection to the X server.

modmap   Specifies the *XModifierKeymap* structure.

The *XSetModifierMapping* function specifies the KeyCodes of the keys (if any) that are to be used as modifiers. If it succeeds, the X server generates a *MappingNotify* event, and *XSetModifierMapping* returns *MappingSuccess*. X permits at most eight modifier keys. If more than eight are specified in the *XModifierKeymap* structure, a *BadLength* error results.

The modifiermap member of the *XModifierKeymap* structure contains eight sets of max_keypermod KeyCodes, one for each modifier in the order *Shift*, *Lock*, *Control*, *Mod1*, *Mod2*, *Mod3*, *Mod4* and *Mod5*. Only non-zero KeyCodes have meaning in each set, and zero KeyCodes are ignored. In addition, all of the non-zero KeyCodes must be in the range specified by min_keycode and max_keycode in the *Display* structure, or a *BadValue* error results.

An X server can impose restrictions on how modifiers can be changed; for example, if certain keys do not generate up transitions in hardware, if auto-repeat cannot be disabled on certain keys, or if multiple modifier keys are not supported. If some such restriction is violated, the status reply is *MappingFailed*, and none of the modifiers are changed. If the new KeyCodes specified for a modifier differ from those currently defined and any (current or new) keys for that modifier are in the logically down state, *XSetModifierMapping* returns *MappingBusy*, and none of the modifiers is changed.

XSetModifierMapping can generate *BadAlloc* and *BadValue* errors.

To obtain the KeyCodes used as modifiers, use *XGetModifierMapping*.

```
XModifierKeymap *XGetModifierMapping(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XGetModifierMapping* function returns a pointer to a newly created *XModifierKeymap* structure that contains the keys being used as modifiers. The structure should be freed after use by calling *XFreeModifiermap*. If only zero values appear in the set for any modifier, that modifier is disabled.

Locales and Internationalized Text Functions

An internationalized application is one which is adaptable to the requirements of different native languages, local customs and character string encodings. The process of adapting the operation to a particular native language, local custom, or string encoding is called localization. A goal of internationalization is to permit localization without program source modifications or recompilation.

Internationalization in X is based on the concept of a *locale*. A locale defines the ''localized'' behaviour of a program at run-time. Locales affect Xlib in its:

- encoding and processing of input method text

- encoding of resource files and values

- encoding and imaging of text strings

- encoding and decoding for inter-client text communication.

Characters from various languages are represented in a computer using an encoding. Different languages have different encodings, and there are even different encodings for the same characters in the same language.

This chapter defines support for localized text imaging and text input, and the locale mechanism which controls all locale-dependent Xlib functions. Sets of functions are provided for multibyte (char *) text as well as wide character (wchar_t) text in the form supported by the host C language environment. The multibyte and wide character functions are equivalent except for the form of the text argument.

The Xlib internationalization functions are not meant to provide support for multilingual applications (mixing multiple languages within a single piece of text), but they make it possible to implement applications that work in limited fashion with more than one language in independent contexts.

14.1   X Locale Management

X supports a one or more of the locales defined by the host environment. On implementations that conform to the ANSI C library, the locale announcement method is *setlocale*. This function configures the locale operation of both the host C library and Xlib. The operation of Xlib is governed by the LC_CTYPE category; this is called the *current locale.* An implementation is permitted to provide implementation-dependent mechanisms for announcing the locale in addition to *setlocale*.

On implementations that do not conform to the ANSI C library, the locale announcement method is Xlib implementation-dependent.

The mechanism by which the semantic operation of Xlib is defined for a specific locale is implementation-dependent.

X is not required to support all the locales supported by the host. To determine if the current locale is supported by X, use *XSupportsLocale*.

```
Bool XSupportsLocale()
```

The *XSupportsLocale* function returns *True* if Xlib functions are capable of operating under the current locale. If it returns *False*, Xlib locale-dependent functions for which the *XLocaleNotSupported* return status is defined will return *XLocaleNotSupported*. Other Xlib locale-dependent routines will operate in the ''C'' locale.

The client is responsible for selecting its locale and X modifiers. Clients should provide a means for the user to override the clients' locale selection at client invocation. Most single-display X clients operate in a single locale for both X and the host processing environment. They will configure the locale by calling three functions: the host locale configuration function, *XSupportsLocale* and *XSetLocaleModifiers*.

The semantics of certain categories of X internationalization capabilities can be configured by setting modifiers. Modifiers are named by implementation-dependent and locale-specific strings. The only standard use for this capability at present is selecting one of several styles of keyboard input method.

To configure Xlib locale modifiers for the current locale, use *XSetLocaleModifiers*.

```
char *XSetLocaleModifiers(modifier_list)
      char *modifier_list;
```

modifier_list   Specifies the modifiers.

XSetLocaleModifiers sets the X modifiers for the current locale setting. The modifier_list argument is a null-terminated string of the form ''{@*category*=*value*}''; that is, having zero or more concatenated ''@*category*=*value*'' entries where *category* is a category name and *value* is the (possibly empty) setting for that category. The values are encoded in the current locale. Category names are restricted to the POSIX Portable Filename Character Set.

The local host X locale modifiers announcer (on POSIX-compliant systems, the XMODIFIERS environment variable) is appended to the modifier_list to provide default values on the local host. If a given category appears more than once in the list, the first setting in the list is used. If a given category is not included in the full modifier list, the category is set to an implementation-dependent default for the current locale. An empty value for a category explicitly specifies the implementation-dependent default.

If the function is successful, it returns a pointer to a string. The contents of the string are such that a subsequent call with that string (in the same locale) will restore the modifiers to the same settings. If modifier_list is a NULL pointer, *XSetLocaleModifiers* also returns a pointer to such a

string, and the current locale modifiers are not changed.

If invalid values are given for one or more modifier categories supported by the locale, a NULL pointer is returned, and none of the current modifiers are changed.

At program startup the modifiers that are in effect are unspecified until the first successful call to set them. Whenever the locale is changed, the modifiers that are in effect become unspecified until the next successful call to set them. Clients should always call *XSetLocaleModifiers* with a non-NULL modifier_list after setting the locale, before they call any locale-dependent Xlib routine.

The only standard modifier category currently defined is ''im'', which identifies the desired input method. The values for input method are not standardized. A single locale may use multiple input methods, switching input method under user control. The modifier may specify the initial input method in effect, or an ordered list of input methods. Multiple input methods may be specified in a single im value string in an implementation-dependent manner.

The returned modifiers string is owned by Xlib and should not be modified or freed by the client. It may be freed by Xlib after the current locale or modifiers is changed. Until freed, it will not be modified by Xlib.

The recommended procedure for clients initializing their locale and modifiers is to obtain locale and modifier announcers separately from one of the following prioritized sources:

- a command line option

- a resource

- the empty string ("").

The first of these that is defined should be used. Note that when a locale command line option or locale resource is defined, the effect should be to set all categories to the specified locale, overriding any category-specific settings in the local host environment.

14.2   Locale and Modifier Dependencies

The internationalized Xlib functions operate in the current locale configured by the host environment and X locale modifiers set by *XSetLocaleModifiers*, or in the locale and modifiers configured at the time some object supplied to the function was created. For each locale-dependent function, the following table describes the locale (and modifiers) dependency:

locale from...	Affects the function...	in the...
	Locale Query/Configuration:	
setlocale	*XSupportsLocale*	locale queried
	XSetLocaleModifiers	locale modified
	Resources:	
setlocale	*XrmGetFileDatabase*	locale of *XrmDatabase*
	XrmGetStringDatabase	
XrmDatabase	*XrmPutFileDatabase*	locale of *XrmDatabase*
	XrmLocaleOfDatabase	
	Setting Standard Properties:	
setlocale	*XmbSetWMProperties*	encoding of supplied/returned text (some WM_ property text in environment locale)
setlocale	*XmbTextPropertyToTextList*	encoding of supplied/returned text
	XwcTextPropertyToTextList	
	XmbTextListToTextProperty	
	XwcTextListToTextProperty	
	Text Input:	
setlocale	*XOpenIM*	XIM input method selection
XIM	*XCreateIC*	XIC input method configuration
	XLocaleOfIM, etc.	queried locale
XIC	*XmbLookupText*	keyboard layout,
	XwcLookupText	encoding of returned text
	Text Drawing:	
setlocale	*XCreateFontSet*	charsets of fonts in *XFontSet*
XFontSet	*XmbDrawText*,	locale of supplied text,
	XwcDrawText, etc.	locale of supplied text,
	XExtentsOfFontSet, etc.	locale-dependent metrics
	XmbTextExtents,	
	XwcTextExtents, etc.	
	Xlib Errors:	
setlocale	*XGetErrorDatabaseText*	locale of error message
	XGetErrorText	

Clients may assume that a locale-encoded text string returned by an X routine can be passed to a C-library routine, or *vice versa*, if the locale is the same at the two calls.

All text strings processed by internationalized Xlib functions are assumed to begin in the initial state of the encoding of the locale, if the encoding is state-dependent.

All Xlib routines behave as if they do not change the current locale or X modifier setting. (This means that if they do change locale or call *XSetLocaleModifiers* with a non-NULL argument, they must save and restore the current state on entry and exit.) Also, Xlib functions on implementations that conform to the ANSI C library do not alter the global state associated with the ANSI C functions *mblen*, *mbtowc*, *wctomb* and *strtok*.

14.3   Creating and Freeing a Font Set

Xlib international text drawing is done using a set of one or more fonts, as needed for the locale of the text. Fonts are loaded according to a list of base font names supplied by the client, and the charsets required by the locale. The *XFontSet* is an opaque type.

To create an international text drawing font set, use *XCreateFontSet*.

```
XFontSet XCreateFontSet(display, base_font_name_list,
        missing_charset_list_return, missing_charset_count_return,
        def_string_return)
    Display *display;
    char *base_font_name_list;
    char ***missing_charset_list_return;
    int *missing_charset_count_return;
    char **def_string_return;
```

display   Specifies the connection to the X server.

base_font_name_list
                Specifies the base font names.

missing_charset_list_return
                Returns the missing charsets.

missing_charset_count_return
                Returns the number of missing charsets.

def_string_return   Returns the string drawn for missing charsets.

The *XCreateFontSet* function creates a font set for the specified display. The font set is bound to the current locale when *XCreateFontSet* is called. The font_set may be used in subsequent calls to obtain font and character information, and to image text in the locale of the font_set.

The base_font_name_list argument is a list of base font names which Xlib uses to load the fonts needed for the locale. The base font names are a comma-separated list. The string is null-terminated, and is assumed to be in the Host Portable Character Encoding; otherwise, the result is implementation-dependent. Whitespace immediately on either side of a separating comma is ignored.

Use of XLFD font names permits Xlib to obtain the fonts needed for a variety of locales from a single locale-independent base font name. The single base font name should name a family of fonts whose members are encoded in the various charsets needed by the locales of interest.

An XLFD base font name can explicitly name a charset needed for the locale. This allows the user to specify an exact font for use with a charset required by a locale, fully controlling the font selection.

If a base font name is not an XLFD name, Xlib will attempt to obtain an XLFD name from the font properties for the font. If this action is successful in obtaining an XLFD name, the *XBaseFontNameListOfFontSet* function will return this XLFD name instead of the client-supplied name.

The following algorithm is used to select the fonts that will be used to display text with the *XFontSet*:

For each font charset required by the locale, the base font name list is searched for the first one of the following cases that names a set of fonts that exist at the server:

1. The first XLFD-conforming base font name that specifies the required charset or a superset of the required charset in its *CharSetRegistry* and *CharSetEncoding* fields. The implementation may use a base font name whose specified charset is a superset of the required charset; for example, an ISO 8859-1 font for an ASCII charset.

2. The first set of one or more XLFD-conforming base font names that specify one or more charsets that can be remapped to support the required charset. The Xlib implementation may recognize various mappings from a required charset to one or more other charsets, and use the fonts for those charsets. For example, JIS Roman is ASCII with tilde and backslash replaced by yen and overbar; Xlib may load an ISO 8859-1 font to support this character set, if a JIS Roman font is not available.

3. The first XLFD-conforming font name, or the first non-XLFD font name for which an XLFD font name can be obtained, combined with the required charset (replacing the *CharSetRegistry* and *CharSetEncoding* fields in the XLFD font name). As in case 1, the implementation may use a charset which is a superset of the required charset.

4. The first font name that can be mapped in some implementation-dependent manner to one or more fonts that support imaging text in the charset.

For example, assume a locale required the charsets:

    ISO 8859-1
    JISX0208.1983
    JISX0201.1976
    GB2312-1980.0

The user could supply a base_font_name_list which explicitly specifies the charsets, ensuring that specific fonts get used if they exist:

    "-JIS-Fixed-Medium-R-Normal--26-180-100-100-C-240-JISX0208.1983-0,\
    -JIS-Fixed-Medium-R-Normal--26-180-100-100-C-120-JISX0201.1976-0,\
    -GB-Fixed-Medium-R-Normal--26-180-100-100-C-240-GB2312-1980.0,\
    -Adobe-Courier-Bold-R-Normal--25-180-75-75-M-150-ISO8859-1"

Or he could supply a base_font_name_list which omits the charsets, letting Xlib select font charsets required for the locale:

    "-JIS-Fixed-Medium-R-Normal--26-180-100-100-C-240,\
    -JIS-Fixed-Medium-R-Normal--26-180-100-100-C-120,\
    -GB-Fixed-Medium-R-Normal--26-180-100-100-C-240,\
    -Adobe-Courier-Bold-R-Normal--25-180-100-100-M-150"

Or he could simply supply a single base font name which allows Xlib to select from all available fonts which meet certain minimum XLFD property requirements:

    "-*-*-*-R-Normal--*-180-100-100-*-*"

If *XCreateFontSet* is unable to create the font set, either because there is insufficient memory or because the current locale is not supported, *XCreateFontSet* returns NULL, missing_charset_list_return is set to NULL, and missing_charset_count_return is set to zero. If fonts exist for all of the charsets required by the current locale, *XCreateFontSet* returns a valid *XFontSet*, missing_charset_list_return is set to NULL, and missing_charset_count_return is set to zero.

If no font exists for one or more of the required charsets, *XCreateFontSet* sets missing_charset_list_return to a list of one or more null-terminated charset names for which no font exists, and sets missing_charset_count_return to the number of missing fonts. The charsets are from the list of the required charsets for the encoding of the locale, and do not include any

charsets to which Xlib may be able to remap a required charset.

If no font exists for any of the required charsets, or if the locale definition in Xlib requires that a font exist for a particular charset and a font is not found for that charset, *XCreateFontSet* returns NULL. Otherwise, *XCreateFontSet* returns a valid *XFontSet* to font_set.

When an Xmb/wc drawing or measuring function is called with an *XFontSet* that has missing charsets, some characters in the locale will not be drawable. If def_string_return is non-NULL, *XCreateFontSet* returns a pointer to a string which represents the glyph(s) which are drawn with this *XFontSet* when the charsets of the available fonts do not include all font glyph(s) required to draw a codepoint. The string does not necessarily consist of valid characters in the current locale and is not necessarily drawn with the fonts loaded for the font set, but the client can draw and measure the ''default glyphs'' by including this string in a string being drawn or measured with the *XFontSet*.

If the string returned to def_string_return is the empty string (""), no glyphs are drawn, and the escapement is zero. The returned string is null-terminated. It is owned by Xlib and should not be modified or freed by the client. It will be freed by a call to *XFreeFontSet* with the associated *XFontSet*. Until freed, its contents will not be modified by Xlib.

The client is responsible for constructing an error message from the missing charset and default string information, and may choose to continue operation in the case that some fonts did not exist.

The returned *XFontSet* and missing charset list should be freed with *XFreeFontSet* and *XFreeStringList*, respectively. The client-supplied base_font_name_list may be freed by the client after calling *XCreateFontSet*.

To obtains a list of *XFontStruct* structures and full font names given an *XFontSet*, use *XFontsOfFontSet*.

```
int XFontsOfFontSet(font_set, font_struct_list_return,
        font_name_list_return)
    XFontSet font_set;
    XFontStruct ***font_struct_list_return;
    char ***font_name_list_return;
```

font_set   Specifies the font set.

font_struct_list_return
                Returns the list of font structs.

font_name_list_return
                Returns the list of font names.

The *XFontsOfFontSet* function returns a list of one or more *XFontStructs* and font names for the fonts used by the Xmb and Xwc layers, for the given font set. A list of pointers to the *XFontStruct* structures is returned to font_struct_list_return. A list of pointers to null-terminated fully specified font name strings in the locale of the font set is returned to font_name_list_return. The font_name_list order corresponds to the font_struct_list order. The number of *XFontStruct* structures and font names is returned as the value of the function.

Because it is not guaranteed that a given character will be imaged using a single font glyph, there is no provision for mapping a character or default string to the font properties, font ID or direction hint for the font for the character. The client may access the *XFontStruct* list to obtain these values for all the fonts currently in use.

It is not required that fonts be loaded from the server at the creation of an *XFontSet*. Xlib may choose to cache font data, loading it only as needed to draw text or compute text dimensions.

Therefore, existence of the per_char metrics in the *XFontStruct* structures in the *XFontStructSet* is undefined. Also, note that all properties in the *XFontStruct* structures are in the STRING encoding.

The *XFontStruct* and font name lists are owned by Xlib and should not be modified or freed by the client. They will be freed by a call to *XFreeFontSet* with the associated *XFontSet*. Until freed, its contents will not be modified by Xlib.

To obtain the base font name list and the selected font name list given an *XFontSet*, use *XBaseFontNameListOfFontSet*.

```
char *XBaseFontNameListOfFontSet(font_set)
      XFontSet font_set;
```

font_set   Specifies the font set.

The *XBaseFontNameListOfFontSet* function returns the original base font name list supplied by the client when the *XFontSet* was created. A null-terminated string containing a list of comma-separated font names is returned as the value of the function. Whitespace may appear immediately on either side of separating commas.

If *XCreateFontSet* obtained an XLFD name from the font properties for the font specified by a non-XLFD base name, the *XBaseFontNameListOfFontSet* function will return the XLFD name instead of the non-XLFD base name.

The base font name list is owned by Xlib and should not be modified or freed by the client. It will be freed by a call to *XFreeFontSet* with the associated *XFontSet*. Until freed, its contents will not be modified by Xlib.

To obtain the locale name given an *XFontSet*, use *XLocaleOfFontSet*.

```
char *XLocaleOfFontSet(font_set)
      XFontSet font_set;
```

font_set   Specifies the font set.

The *XLocaleOfFontSet* function returns the name of the locale bound to the specified *XFontSet*, as a null-terminated string.

The returned locale name string is owned by Xlib and should not be modified or freed by the client. It may be freed by a call to *XFreeFontSet* with the associated *XFontSet*. Until freed, it will not be modified by Xlib.

To free a font set, use *XFreeFontSet*.

```
void XFreeFontSet(display, font_set)
      Display *display;
      XFontSet font_set;
```

display   Specifies the connection to the X server.

font_set   Specifies the font set.

The *XFreeFontSet* function frees the specified font set. The associated base font name list, font name list, *XFontStruct* list and *XFontSetExtents*, if any, are freed.

14.4   Obtaining Font Set Metrics

Metrics for the internationalized text drawing functions are defined in terms of a primary draw direction, which is the default direction in which the character origin advances for each succeeding character in the string. The Xlib interface is currently defined to support only a left-to-right primary draw direction. The drawing origin is the position passed to the drawing function when the text is drawn. The baseline is a line drawn through the drawing origin parallel to the primary draw direction. Character ink is the pixels painted in the foreground color and does not include interline or intercharacter spacing or image text background pixels.

The drawing functions are allowed to implement implicit text directionality control, reversing the order in which characters are rendered along the primary draw direction in response to locale-specific lexical analysis of the string.

Regardless of the character rendering order, the origins of all characters are on the primary draw direction side of the drawing origin. The screen location of a particular character image may be determined with *XmbTextPerCharExtents* or *XwcTextPerCharExtents*.

The drawing functions are allowed to implement context-dependent rendering, where the glyphs drawn for a string are not simply a concatenation of the glyphs that represent each individual character. A string of two characters drawn with *XmbDrawString* may render differently than if the two characters were drawn with separate calls to *XmbDrawString*. If the client appends or inserts a character in a previously drawn string, the client may need to redraw some adjacent characters in order to obtain proper rendering.

To find out about context-dependent rendering, use *XContextDependentDrawing*.

```
Bool XContextDependentDrawing(font_set)
      XFontSet font_set;
```

font_set   Specifies the font set.

The *XContextDependentDrawing* function returns *True* if text drawn with the font_set might include context-dependent drawing.

The drawing functions do not interpret newline, tab or other control characters. The behaviour when non-printing characters other than space are drawn is implementation-dependent. It is the client's responsibility to interpret control characters in a text stream.

The maximum character extents for the fonts that are used by the text drawing layers may be accessed by the *XFontSetExtents* structure:

```
typedef struct {
    XRectangle max_ink_extent;     /* over all drawable characters */
    XRectangle max_logical_extent; /* over all drawable characters */
} XFontSetExtents;
```

The *XRectangles* used to return font set metrics are the usual Xlib screen-oriented *XRectangles*, with x, y giving the upper left corner, and width and height always positive.

The max_ink_extent member gives the maximum extent, over all drawable characters, of the rectangles which bound the character glyph image drawn in the foreground color, relative to a constant origin. See *XmbTextExtents* and *XwcTextExtents* for detailed semantics.

The max_logical_extent member gives the maximum extent, over all drawable characters, of the rectangles which specify minimum spacing to other graphical features, relative to a constant origin. Other graphical features drawn by the client, for example, a border surrounding the text, should not intersect this rectangle. The max_logical_extent member should be used to compute minimum inter-line spacing and the minimum area which must be allowed in a text field to

draw a given number of arbitrary characters.

Due to context-dependent rendering, appending a given character to a string may increase the string's extent by an amount which exceeds the font's max extent:

```
max possible added extent = (max_extent * <total # chars>) −
    prev_string_extent
```

The rectangles for a given character in a string can be obtained from *XmbPerCharExtents* or *XwcPerCharExtents*.

To obtain the maximum extents structure given an *XFontSet*, use *XExtentsOfFontSet*.

```
XFontSetExtents *XExtentsOfFontSet(font_set)
        XFontSet font_set;
```

font_set   Specifies the font set.

The *XExtentsOfFontSet* function returns an *XFontSetExtents* structure for the fonts used by the Xmb and Xwc layers, for the given font set.

The *XFontSetExtents* structure is owned by Xlib and should not be modified or freed by the client. It will be freed by a call to *XFreeFontSet* with the associated *XFontSet*. Until freed, its contents will not be modified by Xlib.

To obtain the escapement in pixels of the specified text as a value, use *XmbTextEscapement* or *XwcTextEscapement*.

```
int XmbTextEscapement(font_set, string, num_bytes)
        XFontSet font_set;
        char *string;
        int num_bytes;


int XwcTextEscapement(font_set, string, num_wchars)
        XFontSet font_set;
        wchar_t *string;
        int num_wchars;
```

font_set   Specifies the font set.

string   Specifies the character string.

num_bytes   Specifies the number of bytes in the string argument.

num_wchars   Specifies the number of characters in the string argument.

The *XmbTextEscapement* and *XwcTextEscapement* functions return the escapement in pixels of the specified string as a value, using the fonts loaded for the specified font set. The escapement is the distance in pixels in the primary draw direction from the drawing origin to the origin of the next character to be drawn, assuming that the rendering of the next character is not dependent on the supplied string.

Regardless of the character rendering order, the escapement is always positive.

To obtain the overall_ink_return and overall_logical_return arguments, the overall bounding box of the string's image, and a logical bounding box, use *XmbTextExtents* or *XwcTextExtents*.

```
int XmbTextExtents(font_set, string, num_bytes, overall_return)
      XFontSet font_set;
      char *string;
      int num_bytes;
      XRectangle *overall_ink_return;
      XRectangle *overall_logical_return;

int XwcTextExtents(font_set, string, num_wchars, overall_return)
      XFontSet font_set;
      wchar_t *string;
      int num_wchars;
      XRectangle *overall_ink_return;
      XRectangle *overall_logical_return;
```

font_set   Specifies the font set.

string   Specifies the character string.

num_bytes   Specifies the number of bytes in the string argument.

num_wchars   Specifies the number of characters in the string argument.

overall_ink_return Returns the overall ink dimensions.

overall_logical_return
                    Returns the overall logical dimensions.

The *XmbTextExtents* and *XwcTextExtents* functions set the components of the specified overall_ink_return and overall_logical_return arguments to the overall bounding box of the string's image, and a logical bounding box for spacing purposes, respectively. They return the value returned by *XmbTextEscapement* or *XwcTextEscapement*. These metrics are relative to the drawing origin of the string, using the fonts loaded for the specified font set.

If the overall_ink_return argument is non-NULL, it is set to the bounding box of the string's character ink. Note that the overall_ink_return for a non-descending horizontally drawn Latin character is conventionally entirely above the baseline, that is, overall_ink_return.height <= -overall_ink_return.y. The overall_ink_return for a nonkerned character is entirely at and to the right of the origin; that is, overall_ink_return.x >= 0. A character consisting of a single pixel at the origin would set overall_ink_return fields y = 0, x = 0, width = 1, height = 1.

If the overall_logical_return argument is non-NULL, it is set to the bounding box which provides minimum spacing to other graphical features for the string. Other graphical features, for example, a border surrounding the text, should not intersect this rectangle.

When the *XFontSet* has missing charsets, metrics for each unavailable character are taken from the default string returned by *XCreateFontSet* so that the metrics represent the text as it will actually be drawn. The behaviour for an invalid codepoint is undefined.

To determine the effective drawing origin for a character in a drawn string, the client should call *XmbTextPerCharExtents* on the entire string, then on the character, and subtract the x values of the returned *XRectangles* for the character. This is useful to redraw portions of a line of text or to justify words, but for context-dependent rendering the client should not assume that it can redraw the character by itself and get the same rendering.

To obtain per-character information for a text string, use *XmbTextPerCharExtents* or *XwcTextPerCharExtents*.

```
Status XmbTextPerCharExtents(font_set, string, num_bytes,
        ink_array_return, logical_array_return, array_size,
        num_chars_return, overall_return)
    XFontSet font_set;
    char *string;
    int num_bytes;
    XRectangle *ink_array_return;
    XRectangle *logical_array_return;
    int array_size;
    int *num_chars_return;
    XRectangle *overall_ink_return;
    XRectangle *overall_logical_return;

Status XwcTextPerCharExtents(font_set, string, num_wchars,
        ink_array_return, logical_array_return, array_size,
        num_chars_return, overall_return)
    XFontSet font_set;
    wchar_t *string;
    int num_wchars;
    XRectangle *ink_array_return;
    XRectangle *logical_array_return;
    int array_size;
    int *num_chars_return;
    XRectangle *overall_ink_return;
    XRectangle *overall_logical_return;
```

font_set　　　　　Specifies the font set.

string　　　　　　Specifies the character string.

num_bytes　　　　Specifies the number of bytes in the string argument.

num_wchars　　　Specifies the number of characters in the string argument.

ink_array_return　Returns the ink dimensions for each character.

logical_array_return
　　　　　　　　　Returns the logical dimensions for each character.

array_size　　　　Specifies the size of ink_array_return and logical_array_return. Note that the caller must pass in arrays of this size.

num_chars_return　Returns the number characters in the string argument.

overall_ink_return　Returns the overall ink extents of the entire string.

overall_logical_return
　　　　　　　　　Returns the overall logical extents of the entire string.

The *XmbTextPerCharExtents* and *XwcTextPerCharExtents* return the text dimensions of each character of the specified text, using the fonts loaded for the specified font set. Each successive element of ink_array_return and logical_array_return is set to the successive character's drawn metrics, relative to the drawing origin of the string, one *XRectangle* for each character in the supplied text string. The number of elements of ink_array_return and logical_array_return that have been set is returned to num_chars_return.

Each element of ink_array_return is set to the bounding box of the corresponding character's drawn foreground color. Each element of logical_array_return is set to the bounding box which

provides minimum spacing to other graphical features for the corresponding character. Other graphical features should not intersect any of the logical_array_return rectangles.

Note that an *XRectangle* represents the effective drawing dimensions of the character, regardless of the number of font glyphs that are used to draw the character, or the direction in which the character is drawn. If multiple characters map to a single character glyph, the dimensions of all the *XRectangles* of those characters are the same.

When the *XFontSet* has missing charsets, metrics for each unavailable character are taken from the default string returned by *XCreateFontSet*, so that the metrics represent the text as it will actually be drawn. The behaviour for an invalid codepoint is undefined.

If the array_size is too small for the number of characters in the supplied text, the functions return zero and num_chars_return is set to the number of rectangles required. Otherwise, the routines return a non-zero value.

If the overall_ink_return or overall_logical_return argument is non-NULL, *XmbTextPerCharExtents* and *XwcTextPerCharExtents* return the maximum extent of the string's metrics to overall_ink_return or overall_logical_return, as returned by *XmbTextExtents* or *XwcTextExtents*.

14.5   Drawing Text Using Font Sets

The routines defined in this section draw text at a specified location in a drawable. They are similar to the functions *XDrawText*, *XDrawString* and *XDrawImageString* except that they work with font sets instead of single fonts, and interpret the text based on the locale of the font set instead of treating the bytes of the string as direct font indexes. See Section 9.6 for details of the use of GCs and possible protocol errors. If a *BadFont* error is generated, characters prior to the offending character may have been drawn.

The text is drawn using the fonts loaded for the specified font set; the font in the GC is ignored, and may be modified by the routines. No validation that all fonts conform to some width rule is performed.

The text functions *XmbDrawText* and *XwcDrawText* use the following structures.

```
typedef struct {
      char *chars;        /* pointer to string */
      int nchars;         /* number of characters */
      int delta;          /* pixel delta between strings */
      XFontSet font_set;  /* fonts, None means don't change */
} XmbTextItem;

typedef struct {
      wchar_t *chars;     /* pointer to wide char string */
      int nchars;         /* number of wide characters */
      int delta;          /* pixel delta between strings */
      XFontSet font_set;  /* fonts, None means don't change */
} XwcTextItem;
```

To draw text using multiple font sets in a given drawable, use *XmbDrawText* or *XwcDrawText*.

```
void XmbDrawText(display, d, gc, x, y,
        items, nitems)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XmbTextItem *items;
      int nitems;

void XwcDrawText(display, d, gc, x, y,
        items, nitems)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XwcTextItem *items;
      int nitems;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

x
y   Specify the x and y coordinates.

items   Specifies an array of text items.

nitems   Specifies the number of text items in the array.

XmbDrawText and *XwcDrawText* allow complex spacing and font set shifts between text strings. Each text item is processed in turn, with the origin of a text element advanced in the primary draw direction by the escapement of the previous text item. A text item delta specifies an additional escapement of the text item drawing origin in the primary draw direction. A font_set member other than *None* in an item causes the font set to be used for this and subsequent text items in the text_items list. Leading text items with font_set member set to *None* will not be drawn.

XmbDrawText and *XwcDrawText* do not perform any context-dependent rendering between text segments. Clients may compute the drawing metrics by passing each text segment to *XmbTextExtents* and *XwcTextExtents*, or *XmbTextPerCharExtents* and *XwcTextPerCharExtents*. When the *XFontSet* has missing charsets, each unavailable character is drawn with the default string returned by *XCreateFontSet*. The behaviour for an invalid codepoint is undefined.

To draw text using a single font set in a given drawable, use *XmbDrawString* or *XwcDrawString*.

```
void XmbDrawString(display, d, font_set, gc, x,
        y, string, num_bytes)
    Display *display;
    Drawable d;
    XFontSet font_set;
    GC gc;
    int x, y;
    char *string;
    int num_bytes;


void XwcDrawString(display, d, font_set, gc, x,
        y, string, num_wchars)
    Display *display;
    Drawable d;
    XFontSet font_set;
    GC gc;
    int x, y;
    wchar_t *string;
    int num_wchars;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

font_set   Specifies the font set.

gc   Specifies the GC.

x
y   Specify the x and y coordinates.

string   Specifies the character string.

num_bytes   Specifies the number of bytes in the string argument.

num_wchars   Specifies the number of characters in the string argument.

XmbDrawString and *XwcDrawString* draw the specified text with the foreground pixel. When the *XFontSet* has missing charsets, each unavailable character is drawn with the default string

returned by *XCreateFontSet*. The behaviour for an invalid codepoint is undefined.

To draw image text using a single font set in a given drawable, use *XmbDrawImageString* or *XwcDrawImageString*.

```
void XmbDrawImageString(display, d, font_set, gc,
        x, y, string, num_bytes)
    Display *display;
    Drawable d;
    XFontSet font_set;
    GC gc;
    int x, y;
    char *string;
    int num_bytes;

void XwcDrawImageString(display, d, font_set, gc,
        x, y, string, num_wchars)
    Display *display;
    Drawable d;
    XFontSet font_set;
    GC gc;
    int x, y;
    wchar_t *string;
    int num_wchars;
```

display	Specifies the connection to the X server.
d	Specifies the drawable.
font_set	Specifies the font set.
gc	Specifies the GC.
x	
y	Specify the x and y coordinates.
string	Specifies the character string.
num_bytes	Specifies the number of bytes in the string argument.
num_wchars	Specifies the number of characters in the string argument.

XmbDrawImageString and *XwcDrawImageString* fill a destination rectangle with the background pixel defined in the GC and then paint the text with the foreground pixel. The filled rectangle is the rectangle returned to overall_logical_return by *XmbTextExtents* or *XwcTextExtents* for the same text and *XFontSet*.

When the *XFontSet* has missing charsets, each unavailable character is drawn with the default string returned by *XCreateFontSet*. The behaviour for an invalid codepoint is undefined.

14.6   Input Method Overview

This section provides definitions for terms and concepts used for internationalized text input and a brief overview of the intended use of the mechanisms provided by Xlib.

A large number of languages in the world use alphabets consisting of a small set of symbols (letters) to form words. To enter text into a computer in an alphabetic language, a user usually has a keyboard on which there exists key symbols corresponding to the alphabet. Sometimes, a few characters of an alphabetic language are missing on the keyboard. Many computer users, who speak a Latin alphabet based language only have an English-based keyboard. They need to hit a combination of keystrokes in order to enter a character that does not exist directly on the keyboard. A number of algorithms have been developed for entering such characters, known as European input methods, the compose input method, or the dead-keys input method.

Japanese is an example of a language with a phonetic symbol set, where each symbol represents a specific sound. There are two phonetic symbol sets in Japanese: Katakana and Hiragana. In general, Katakana is used for words that are of foreign origin, and hiragana for writing native Japanese words. Collectively, the two systems are called Kana. Each set consists of 48 characters.

Korean also has a phonetic symbol set, called Hangul. Each of the 24 basic phonetic symbols (14 consonants and 10 vowels) represents a specific sound. A syllable is composed of two or three parts: the initial consonants, the vowels and the optional last consonants. With Hangul, syllables can be treated as the basic units on which text processing is done. For example, a delete operation may work on a phonetic symbol or a syllable. Korean code sets include several thousands of these syllables. A user types the phonetic symbols that make up the syllables of the words to be entered. The display may change as each phonetic symbol is entered. For example, when the second phonetic symbol of a syllable is entered, the first phonetic symbol may change its shape and size. Likewise, when the third phonetic symbol is entered, the first two phonetic symbols may change their shape and size.

Not all languages rely solely on alphabetic or phonetic systems. Some languages, including Japanese and Korean, employ an ideographic writing system. In an ideographic system, rather than taking a small set of symbols and combining them in different ways to create words, each word consists of one unique symbol (or, occasionally, several symbols). The number of symbols may be very large: approximately 50,000 have been identified in Hanzi, the Chinese ideographic system.

There are two major aspects of ideographic systems for their computer usage. First, the standard computer character sets in Japan, China and Korea include roughly 8,000 characters, while sets in Taiwan have between 15,000 and 30,000 characters, which make it necessary to use more than one byte to represent a character. Second, it obviously is impractical to have a keyboard that includes all of a given language's ideographic symbols. Therefore a mechanism is required for entering characters so that a keyboard with a reasonable number of keys can be used. Those input methods are usually based on phonetics, but there also exist methods based on the graphical properties of characters.

In Japan, both Kana and Kanji are used. In Korea, Hangul and sometimes Hanja are used. Now consider entering ideographs in Japan, Korea, China and Taiwan.

In Japan, either Kana or English characters are typed and then a region is selected (sometimes automatically) for conversion to Kanji. Several Kanji characters may have the same phonetic representation. If that is the case with the string entered, a menu of characters is presented and the user must choose the appropriate one. If no choice is necessary or a preference has been established, the input method does the substitution directly. When Latin characters are converted to Kana or Kanji, it is called a romaji conversion.

Stamp:XXXXXXXXXXXXXXXXXXXXXXXX

In Korea, it is usually acceptable to keep Korean text in Hangul form, but some people may choose to write Hanja-originated words in Hanja rather than in Hangul. To change Hangul to Hanja, a region is selected for conversion and then the same basic method as described for Japanese is followed.

Probably because there are well-accepted phonetic writing systems for Japanese and Korean, computer input methods in these countries for entering ideographs are fairly standard. Keyboard keys have both English characters and phonetic symbols engraved on them, and the user can switch between the two sets.

The situation is different for Chinese. While there is a phonetic system called Pinyin promoted by authorities, there is no consensus for entering Chinese text. Some vendors use a phonetic decomposition (Pinyin or another), others use ideographic decomposition of Chinese words, with various implementations and keyboard layouts. There are about 16 known methods, none of which is a clear standard.

Also, there are actually two ideographic sets used: Traditional Chinese (the original written Chinese) and Simplified Chinese. Several years ago, the People's Republic Of China launched a campaign to simplify some ideographic characters and eliminate redundancies altogether. Under the plan, characters would be streamlined every five years. Characters have been revised several times now, resulting in the smaller, simpler set that makes up Simplified Chinese.

14.6.1   Input Method Architecture

As shown in the previous section, there are many different input methods in use today, each varying with language, culture and history. A common feature of many input methods is that the user may type multiple keystrokes in order to compose a single character (or set of characters). The process of composing characters from keystrokes is called *preediting*. It may require complex algorithms and large dictionaries involving substantial computer resources.

Input methods may require one or more areas in which to show the feedback of the actual keystrokes, to propose disambiguation to the user, to list dictionaries, and so on. The input method areas of concern are as follows.

- The *Status* area is intended to be a logical extension of the LEDs that exist on the physical keyboard. It is a window which is intended to present the internal state of the input method that is critical to the user. The status area may consist of text data and bitmaps or some combination.

- The *Preedit* area is intended to display the intermediate text for those languages that are composing prior to the client handling the data.

- The *Auxiliary* area is used for pop-up menus and customizing dialogs that may be required for an input method. There may be multiple Auxiliary areas for any input method. Auxiliary areas are managed by the input method independent of the client. Auxiliary areas are assumed to be a separate dialog which is maintained by the input method.

There are various user interaction styles used for preediting. The ones supported by Xlib are as follows.

- For *on-the-spot* input methods, preediting data will be displayed directly in the application window. Application data is moved to allow preedit data to be displayed at the point of insertion.

- *Over-the-spot* preediting means that the data is displayed in a preedit window that is placed over the point of insertion.

- *Off-the-spot* preediting means that the preedit window is inside the application window but not at the point of insertion. Often, this type of window is placed at the bottom of the application window.

- *Root-window* preediting refers to input methods that use a preedit window that is the child of *RootWindow*.

It would require a lot of computing resources if portable applications had to include input methods for all the languages in the world. To avoid this, a goal of the Xlib design is to allow an application to communicate with an input method placed in a separate process. Such a process is called an *input server*. The server to which the application should connect is dependent on the environment when the application is started up: what is the user language, the actual encoding to be used for it. The input method connection is said to be *locale-dependent*. It is also user-dependent: for a given language, the user can choose to some extent the user interface style of input method (if choice is possible among several).

Using an input server implies communication overhead, but applications can be migrated without relinking. Input methods can be implemented either as a stub communicating to an input server or as a local library.

An input method may be based on a *front-end* or a *back-end* architecture. In front-end, there are two separate connections to the X server: keystrokes go directly from X server to the input method on one connection, other events to the regular client connection. The input method is then acting as a filter, and sends composed strings to the client. Front-end requires synchronization between the two connections to avoid lost key events or locking issues.

In back-end, a single X server connection is used. A dispatching mechanism must decide on this channel to delegate appropriate keystrokes to the input method. For instance, it may retain a Help keystroke for its own purpose. In the case where the input method is a separate process (that is, a server), there must be a special communication protocol between the back-end client and the input server.

Front-end introduces synchronization issues and filtering mechanism for non-character keystrokes (Functions, Help, and so on). Back-end sometimes implies more communication overhead and more process switching. If all three processes are running on a single workstation (X server, input server, client), there are two process switches for each keystroke in back-end, but there is only one in frontend.

The abstraction used by a client to communicate with an input method is an opaque data structure represented by the *XIM* data type. This data structure is returned by the *XOpenIM* function, which opens an input method on a given display. Subsequent operations on this data structure encapsulate all communication between client and input method. There is no need for an X client to use any networking library or natural language package in order to use an input method.

A single input server may be used for one or more languages, supporting one or more encoding schemes. But the strings returned from an input method will always be encoded in the (single) locale associated with *XIM* object.

14.6.2   Input Contexts

Xlib provides the ability to manage a multithreaded state for text input. A client may be using multiple windows, each window with multiple text entry areas, and the user possibly switching among them at any time. The abstraction for representing state of a particular input thread is called an *input context*. The Xlib representation of an input context is an *XIC*.

An input context is the abstraction retaining the state, properties and semantics of communication between a client and an input method. An input context is a combination of an input method, a locale specifying the encoding of the character strings to be returned, a client window, internal state information and various layout or appearance characteristics. The input context concept somewhat matches for input the graphics context abstraction defined for graphics output.

One input context belongs to exactly one input method. Different input contexts may be associated with the same input method, possibly with the same client window. An *XIC* is created with the *XCreateIC* function, providing an *XIM* argument, affiliating the input context to the input method for its lifetime. When an input method is closed with *XCloseIM*, all of its affiliated input contexts should not be used any more (and should preferably be destroyed before closing the input method).

Considering the example of a client window with multiple text entry areas, the application programmer could for example choose to implement:

- As many input contexts are created as text entry areas and the client will get the input accumulated on each context each time it will lookup that context.

- A single context is created for a top level window in the application. If such window contains several text entry areas, each time the user moves to another text entry area, the client has to indicate changes in the context.

A range of choices can be made by application designers to use either a single or multiple input contexts, according to the needs of their application.

14.6.3   Getting Keyboard Input

In order to obtain characters from an input method a client must call the function *XmbLookupString* or *XwcLookupString* with an input context created from that input method. Both a locale and display are bound to an input method when it is opened, and an input context inherits this locale and display. Any strings returned by *XmbLookupString* or *XwcLookupString* will be encoded in that locale.

14.6.4   Focus Management

For each text entry area in which the *XmbLookupString* or *XwcLookupString* routines are used there will be an associated input context.

When the application focus moves to a text entry area, the application must set the input context focus to the input context associated with that area. The input context focus is set by calling *XSetICFocus* with the appropriate input context.

Also, when the application focus moves out of a text entry area, the application should unset the focus for the associated input context by calling *XUnsetICFocus*. As an optimization, if *XSetICFocus* is called successively on two different input contexts, setting the focus on the second will automatically unset the focus on the first.

Note that in order to set and unset the input context focus correctly, it will be necessary to track application-level focus changes. Such focus changes do not necessarily correspond to X server

focus changes.

If a single input context is being used to do input for multiple text entry areas, it will also be necessary to set the focus window of the input context whenever the focus window changes (see *XNFocusWindow* under *XSetICValues*).

14.6.5 Geometry Management

In most input method architectures (on-the-spot being the notable exception), the input method will perform the display of its own data. In order to provide better visual locality, it is often desirable to have the input method areas embedded within a client. In order to do this the client may need to allocate space for an input method. Xlib provides support that allows the size and position of input method areas to be provided by a client. The input method areas that are supported for geometry management are the Status area and the Preedit area.

The fundamental concept on which geometry management for input method windows is based is the proper division of responsibilities between the client (or toolkit) and the input method. The division of responsibilities is as follows:

- The client is responsible for the geometry of the input method window.

- The input method is responsible for the contents of the input method window.

An input method is able to suggest a size to the client, but it cannot suggest a placement. Also the input method can only suggest a size. It does not determine the size, and it must accept the size it is given.

Before a client provides geometry management for an input method, it must determine if geometry management is needed. The input method indicates the need for geometry management by setting *XIMPreeditArea* or *XIMStatusArea* in its *XIMStyles* value returned by *XGetIMValues*. When a client has decided that it will provide geometry management for an input method, it indicates that decision by setting the *XNInputStyle* value in the *XIC*.

After a client has established with the input method that it will will do geometry management, the client must negotiate the geometry with the input method. The geometry is negotiated by the following steps.

- The client suggests an area to the input method by setting the *XNAreaNeeded* value for that area. If the client has no constraints for the input method it either will not suggest an area or will set the width and height to zero. Otherwise it will set one of the values.

- The client will get the XIC value *XNAreaNeeded*. The input method will return its suggested size in this value. The input method should pay attention to any constraints suggested by the client.

- The client sets the XIC value *XNArea* to inform the input method of the geometry of its window. The client should try to honour the geometry requested by the input method. The input method must accept this geometry.

Clients doing geometry management must be aware that setting other IC values may affect the geometry desired by an input method. For example, *XNFontSet* and *XNLineSpacing* may change the geometry desired by the the input method.

The table of XIC values (see Section 14.10) indicates the values that can cause the desired geometry to change when they are set. It is the responsibility of the client to renegotiate the geometry of the input method window when it is needed.

In addition, a geometry management callback is provided by which an input method can initiate a geometry change.

14.6.6   Event Filtering

A filtering mechanism is provided to allow input methods to capture X events transparently to clients. It is expected that toolkits (or clients) using *XmbLookupString* or *XwcLookupString* will call this filter at some point in the event processing mechanism to make sure that events needed by an input method can be filtered by that input method.

If there were no filter, a client could receive and discard events that are necessary for the proper functioning of an input method. The following provides a few examples of such events:

- Expose events on preedit window in local mode.

- Events may be used by an input method to communicate with an input server. Such input server protocol related events have to be intercepted if one does not want to disturb client code.

- Key events can be sent to a filter before they are bound to translations such as Xt provides.

Clients are expected to get the XIC value *XNFilterEvents* and augment the event mask for the client window with that event mask. This mask may be zero.

14.6.7   Callbacks

When an on-the-spot input method is implemented, only the client can insert or delete preedit data in place and possibly scroll existing text. This means the echo of the keystrokes has to be achieved by the client itself, tightly coupled with the input method logic.

When a keystroke is entered, the client calls *XmbLookupString* or *XwcLookupString*. At this point, in the on-the-spot case, the echo of the keystroke in the preedit has not yet been done. Before returning to the client logic that handles the input characters, the lookup function must call the echoing logic for inserting the new keystroke. If the keystrokes entered so far make up a character, the keystrokes entered need to be deleted, and the composed character will be returned. Hence, what happens is that, while being called by client code, input method logic has to call back to the client before it returns. The client code, that is, a callback routine, is called from the input method logic.

There are a number of cases where the input method logic has to call back the client. Each of those cases is associated with a well-defined callback action. It is possible for the client to specify, for each input context, what callback is to be called for each action.

There are also callbacks provided for feedback of status information and a callback to initiate a geometry request for an input method.

14.7   Variable Argument Lists

Several input method functions have arguments which conform to ANSI C variable argument list calling convention. Each function denoted with a ''...'' argument takes a variable length list of name and value pairs where each name is a string and each value is of type *XPointer*. The end of the list is identified by a name argument containing NULL.

A variable length argument list may contain a nested list. If the name *XVaNestedList* is specified in place of an argument name, then the following value is interpreted as a *XVaNestedList* value which specifies a list of values logically inserted into the original list at the point of declaration. The end of a nested list is identified with a NULL.

To allocate a nested variable argument list dynamically, use *XVaCreateNestedList*.

```
typedef void * XVaNestedList;

XVaNestedList XVaCreateNestedList(dummy, ...)
      int dummy;
```

dummy   Unused argument (required by ANSI C).

...   Specifies the variable length argument list.

The *XVaCreateNestedList* function allocates memory and copies its arguments into a single list pointer which may be used as value for arguments requiring a list value. Any entries are copied as specified. Data passed by reference is not copied; the caller must ensure data remains valid for the lifetime of the nested list. The list should be freed using *XFree* when it is no longer needed.

X/Open CAE Specification

14.8   Input Method Functions

To open a connection, use *XOpenIM*.

```
XIM XOpenIM(display, db, res_name, res_class)
      Display *display;
      XrmDataBase db;
      char *res_name;
      char *res_class;
```

display   Specifies the connection to the X server.

db   Specifies a pointer to the resource database.

res_name   Specifies the full resource name of the application.

res_class   Specifies the full class name of the application.

The *XOpenIM* function opens an input method, matching the current locale and modifiers specification. Current locale and modifiers are bound to the input method at opening time. The locale associated with an input method cannot be changed dynamically. This implies the strings returned by *XmbLookupString* or *XwcLookupString*, for any input context affiliated with a given input method, will be encoded in the locale current at the time input method is opened.

The specific input method to which this call will be routed is identified on the basis of the current locale. *XOpenIM* will identify a default input method corresponding to the current locale. That default can be modified using *XSetLocaleModifiers* for the input method modifier.

The db argument is the resource database to be used by the input method for looking up resources that are private to the input method. It is not intended that this database be used to look up values that can be set as IC values in an input context. If db is NULL, no data base is passed to the input method.

The res_name and res_class arguments specify the resource name and class of the application. They are intended to be used as prefixes by the input method when looking up resources that are common to all input contexts that may be created for this input method. The characters used for resource names and classes must be in the X portable character set. The resources looked up are not fully specified if res_name or res_class is NULL.

The res_name and res_class arguments are not assumed to exist beyond the call to *XOpenIM*. The specified resource database is assumed to exist for the lifetime of the input method.

XOpenIM returns NULL if no input method could be opened.

To close a connection, use *XCloseIM*.

```
Status XCloseIM(im)
      XIM im;
```

im   Specifies the input method.

The *XCloseIM* function closes the specified input method.

To query an input method, use *XGetIMValues*.

```
char * XGetIMValues(im, ...)
      XIM im;
```

im   Specifies the input method.

...   Specifies the variable length argument list to get XIM values.

The *XGetIMValues* function presents a variable argument list programming interface for querying properties or features of the specified input method. This function returns NULL if it succeeds; otherwise, it returns the name of the first argument that could not be obtained.

Only one standard argument is defined by Xlib: *XNQueryInputStyle*, which must be used to query about input styles supported by the input method.

A client should always query the input method to determine which styles are supported. The client should then find an input style it is capable of supporting.

If the client cannot find an input style that it can support it should negotiate with the user the continuation of the program (exit, choose another input method, and so on).

The argument value must be a pointer to a location where the returned value will be stored. The returned value is a pointer to a structure of type *XIMStyles*. Clients are responsible for freeing the *XIMStyles* data structure. To do so, use *XFree*.

The *XIMStyles* structure is defined as follows.

```
typedef unsigned long XIMStyle;

#define XIMPreeditArea 0x0001L
#define XIMPreeditCallbacks 0x0002L
#define XIMPreeditPosition 0x0004L
#define XIMPreeditNothing 0x0008L
#define XIMPreeditNone 0x0010L
#define XIMStatusArea 0x0100L
#define XIMStatusCallbacks 0x0200L
#define XIMStatusNothing 0x0400L
#define XIMStatusNone 0x0800L

typedef struct {
      unsigned short count_styles;
      XIMStyle * supported_styles;
} XIMStyles;
```

An *XIMStyles* structure contains in its field count_styles, the number of input styles supported. This is also the size of the array in the field supported_styles.

The supported styles is a list of bit mask combinations, which indicate the combination of styles for each of the areas supported. These areas are described below. Each element in the list should select one of the bit mask values for each area. The list describes the complete set of combinations supported. Only these combinations are supported by the input method.

The Preedit category defines what type of support is provided by the input method for preedit information:

XIMPreeditArea   If chosen, the input method would require the client to provide some area values for it to do its preediting. Refer to XIC values *XNArea* and *XNAreaNeeded*.

XIMPreeditPosition

If chosen, the input method would require the client to provide positional values. Refer to XIC values *XNSpotLocation* and *XNFocusWindow*.

XIMPreeditCallbacks

If chosen, the input method would require the client to define the set of preedit callbacks. Refer to XIC values *XNPreeditStartCallback*, *XNPreeditDoneCallback*, *XNPreeditDrawCallback* and *XNPreeditCaretCallback*.

XIMPreeditNothing
> If chosen, the input method can function without any Preedit values.

XIMPreeditNone The input method does not provide any Preedit feedback. Any Preedit value is ignored. This style is mutually exclusive with the other Preedit styles.

The Status category defines what type of support is provided by the input method for status information:

XIMStatusArea The input method requires the client to provide some area values for it to do its Status feedback. See *XNArea* and *XNAreaNeeded*.

XIMStatusCallbacks
> The input method requires the client to define the set of status callbacks; *XNStatusStartCallback*, *XNStatusDoneCallback* and *XNStatusDrawCallback*.

XIMStatusNothing
> The input method can function without any Status values.

XIMStatusNone The input method does not provide any Status feedback. If chosen, any Status value is ignored. This style is mutually exclusive with the other Status styles.

To obtain the display associated with an input method, use *XDisplayOfIM*.

```
Display * XDisplayOfIM(im)
      XIM im;
```

im   Specifies the input method.

The *XDisplayOfIM* function returns the display associated with the specified input method.

To get the locale associated with an input method, use *XLocaleOfIM*.

```
char * XLocaleOfIM(im)
      XIM im;
```

im   Specifies the input method.

The *XLocaleOfIM* returns the locale associated with the specified input method.

14.9   Input Context Functions

An input context is an abstraction that is used to contain both the data required (if any) by an input method and the information required to display that data. There may be multiple input contexts for one input method. The programming interfaces for creating, reading or modifying an input context use a variable argument list. The name elements of the argument lists are referred to as XIC values. It is intended that input methods be controlled by these XIC values. As new XIC values are created they should be registered with the X Consortium.

To create an input context use *XCreateIC*.

```
XIC XCreateIC(im, ...)
      XIM im;
```

im   Specifies the input method.

...   Specifies the variable length argument list to set XIC values.

The *XCreateIC* function creates a context within the specified input method.

Some of the arguments are mandatory at creation time, and the input context will not be created if they are not provided. Those arguments are the input style and the set of text callbacks (if the input style selected requires callbacks). All other input context values can be set later.

XCreateIC returns a NULL value if no input context could be created. A NULL value could be returned for any of the following reasons:

- A required argument was not set.

- A read-only argument was set (for example, *XNFilterEvents*).

- The argument name is not recognized.

- The input method encountered an input method implementation-dependent error.

XCreateIC can generate *BadAtom*, *BadColor*, *BadPixmap* and *BadWindow* errors.

To destroy an input context, use *XDestroyIC*.

```
void XDestroyIC(ic)
      XIC ic;
```

ic   Specifies the input context.

XDestroyIC destroys the specified input context.

To communicate to and synchronize with input method for any changes in keyboard focus from the client side, use *XSetICFocus* and *XUnsetICFocus*.

```
void XSetICFocus(ic)
      XIC ic;
```

ic   Specifies the input context.

The *XSetICFocus* function allows a client to notify an input method that the focus window attached to the specified input context has received keyboard focus. The input method should take action to provide appropriate feedback. Complete feedback specification is a matter of user interface policy.

```
void XUnsetICFocus(ic)
      XIC ic;
```

ic   Specifies the input context.

The *XUnsetICFocus* function allows a client to notify an input method that the specified input context has lost the keyboard focus and that no more input is expected on the focus window attached to that input context. The input method should take action to provide appropriate feedback. Complete feedback specification is a matter of user interface policy.

To reset the state of an input context to initial state, use *XmbResetIC* or *XwcResetIC*.

```
char * XmbResetIC(ic)
      XIC ic;


wchar_t * XwcResetIC(ic)
      XIC ic;
```

ic   Specifies the input context.

The *XmbResetIC* and *XwcResetIC* functions reset input context to initial state. Any input pending on that context is deleted. Input method is required to clear preedit area, if any, and update status accordingly. Calling *XmbResetIC* or *XwcResetIC* does not change the focus.

The return value of *XmbResetIC* is its current preedit string as a multibyte string. The return value of *XwcResetIC* is its current preedit string as a wide character string. It is input method implementation-dependent whether these routines return a non-NULL string or NULL.

The client should free the returned string by calling *XFree*.

To get the input method associated with an input context, use *XIMOfIC*.

```
XIM XIMOfIC(ic)
      XIC ic;
```

ic   Specifies the input context.

The *XIMOfIC* function returns the input method associated with the specified input context.

Xlib provides two functions for setting and reading XIC values, respectively: *XSetICValues* and *XGetICValues*. Both functions have a variable length argument list. In that argument list, any XIC value's name must be denoted with a character string using the X Portable Character Set.

To set XIC values, use *XSetICValues*.

```
char * XSetICValues(ic, ...)
      XIC ic;
```

ic   Specifies the input context.

...   Specifies the variable length argument list to set XIC values.

The *XSetICValues* function returns NULL if no error occurred; otherwise, it returns the name of the first argument that could not be set. An argument could be not set for any of the following reasons:

- A read-only argument was set (for example, *XNFilterEvents*).

- The argument name is not recognized.

- The input method encountered an input method implementation-dependent error.

Each value to be set must be an appropriate datum, matching the data type imposed by the semantics of the argument.

XSetICValues can generate *BadAtom*, *BadColor*, *BadCursor*, *BadPixmap* and *BadWindow* errors.

To obtain XIC values, use *XGetICValues*.

```
char * XGetICValues(ic, ...)
      XIC ic;
```

ic   Specifies the input context.

...   Specifies the variable length argument list to get XIC values.

The *XGetICValues* function returns NULL if no error occurred; otherwise, it returns the name of the first argument that could not be obtained. An argument could be not obtained for any of the following reasons:

- The argument name is not recognized.

- The input method encountered an implementation-dependent error.

Each argument value (following a name) must point to a location where the value is to be stored. *XGetICValues* allocates memory to store the values, and client is responsible for freeing each value by calling *XFree*.

14.10 XIC Value Arguments

The following tables describe how XIC values are interpreted by an input method depending on the input style chosen by the user.

The first column lists the XIC values. The second column indicates which values are involved in affecting, negotiating, and setting the geometry of the input method windows. The subentries under the third column indicate the different input styles that are supported. Each of these columns indicates how each of the XIC values are treated by that input style.

The following *Keys* apply to these tables.

Keys	Explanation
C	This value must be set with *XCreateIC*.
D	This value may be set using *XCreateIC*. If it is not set, a default is provided.
G	This value may be read using *XGetICValues*.
GN	This value may cause geometry negotiation when its value is set by means of *XCreateIC* or *XSetICValues*.
GR	This value will be the response of the input method when any GN value is changed.
GS	This value will cause the geometry of the input method window to be set.
O	This value must be set once and only once. It need not be set at create time.
S	This value may be set with *XSetICValues*.
ignored	This value is ignored by the input method for the given input style.

XIC Value	Geometry Management	Input Style				
		Preedit Callback	**Preedit Position**	**Preedit Area**	**Preedit Nothing**	**Preedit None**
Input Style		C-G	C-G	C-G	C-G	C-G
Client Window		O-G	O-G	O-G	ignored	ignored
Focus Window	GN	D-S-G	D-S-G	D-S-G	D-S-G	ignored
Resource Name		ignored	D-S-G	D-S-G	D-S-G	ignored
Resource Class		ignored	D-S-G	D-S-G	D-S-G	ignored
Geometry Callback		ignored	ignored	D-S-G	ignored	ignored
FilterEvents		G	G	G	G	ignored
Preedit						
Area	GS	ignored	D-S-G	D-S-G	ignored	ignored
AreaNeeded	GN-GR	ignored	ignored	S-G	ignored	ignored
SpotLocation		ignored	C-S-G	ignored	ignored	ignored
Colormap		ignored	D-S-G	D-S-G	D-S-G	ignored
Foreground		ignored	D-S-G	D-S-G	D-S-G	ignored
Background		ignored	D-S-G	D-S-G	D-S-G	ignored
Background Pixmap		ignored	D-S-G	D-S-G	D-S-G	ignored
FontSet	GN	ignored	C-S-G	C-S-G	D-S-G	ignored
LineSpacing	GN	ignored	D-S-G	D-S-G	D-S-G	ignored
Cursor		ignored	D-S-G	D-S-G	D-S-G	ignored
Preedit Callbacks		C-S-G	ignored	ignored	ignored	ignored

		Input Style			
XIC Value	**Geometry Management**	**Status Callback**	**Status Area**	**Status Nothing**	**Status None**
Input Style		C-G	C-G	C-G	C-G
Client Window		O-G	O-G	O-G	ignored
Focus Window	GN	D-S-G	D-S-G	D-S-G	ignored
Resource Name		ignored	D-S-G	D-S-G	ignored
Resource Class		ignored	D-S-G	D-S-G	ignored
Geometry Callback		ignored	D-S-G	ignored	ignored
FilterEvents		G	G	G	G
Status					
Area	GS	ignored	D-S-G	ignored	ignored
AreaNeeded	GN-GR	ignored	S-G	ignored	ignored
Colormap		ignored	D-S-G	D-S-G	ignored
Foreground		ignored	D-S-G	D-S-G	ignored
Background		ignored	D-S-G	D-S-G	ignored
Background Pixmap		ignored	D-S-G	D-S-G	ignored
FontSet	GN	ignored	C-S-G	D-S-G	ignored
LineSpacing	GN	ignored	D-S-G	D-S-G	ignored
Cursor		ignored	D-S-G	D-S-G	ignored
Status Callbacks		C-S-G	ignored	ignored	ignored

14.10.1 Input Style

The *XNInputStyle* argument specifies the input style to be used. The value of this argument must be one of the values returned by the *XGetIMValues* function with the *XNQueryInputStyle* argument specified in the supported_styles list.

Note that this argument must be set at creation time and cannot be changed.

14.10.2 Client Window

The *XNClientWindow* argument specifies to the input method the client window in which the input method can display data or create subwindows. Geometry values for input method areas are given with respect to the client window. Dynamic change of client window is not supported. Note that this argument may be set only once and should be set before any input is done using this input context. If it is not set the input method may not operate correctly.

If an attempt is made to set this value a second time with *XSetICValues*, the string *XNClientWindow* will be returned by *XSetICValues*, and the client window will not be changed.

If the client window is not a valid window ID on the display attached to the input method, a *BadWindow* error can be generated when this value is used by the input method.

14.10.3 Focus Window

The *XNFocusWindow* argument specifies the focus window. The primary purpose of the *XNFocusWindow* is to identify the window that will receive the key event when input is composed. In addition, the input method may possibly affect the focus window as follows:

- select events on it

- send events to it

- modify its properties

- grab keyboard within that window.

The value associated to the argument must be of type *Window*. If the focus window is not a valid window ID on the display attached to the input method, a *BadWindow* error can be generated when this value is used by the input method.

When this XIC value is left unspecified, the input method will default focus window to client window.

14.10.4 Resource Name and Class

The *XNResourceName* and *XNResourceClass* arguments are strings that specify the full name and class used by the client to obtain resources for the client window. These values should be used as prefixes for name and class when looking up resources that may vary according to the input context. If these values are not set, the resources will not be fully specified.

It is not intended that values which can be set as XIC values be set as resources.

14.10.5 Geometry Callback

The *XNGeometryCallback* argument is a structure of type *XIMCallback* (see **Preedit and Status Callbacks**).

The *XNGeometryCallback* argument specifies the geometry callback which a client can set. This callback is not required for correct operation of either an input method or a client. It can be set for a client whose user interface policy permits an input method to request the dynamic change of that input methods window. An input method that does dynamic change will need to filter any events that it uses to initiate the change.

14.10.6 Filter Events

The *XNFilterEvents* argument returns the event mask that an input methods needs to have selected for. The client is expected to augment its own event mask for the client window with this one.

Note that this argument is read-only, is set by the input method at create time, and is never changed.

Note also that the type of this argument is ''unsigned long''. Setting this value will cause an error.

14.10.7 Preedit and Status Attributes

The *XNPreeditAttributes* and *XNStatusAttributes* arguments specify to input method the attributes to be used for the Preedit and Status areas, if any. Those attributes are passed to *XSetICValues* or *XGetICValues* as a nested variable length list. The names to be used in these lists are as described in the following sections.

Area

The value of the *XNArea* argument must be a pointer to a structure of type *XRectangle.* The interpretation of the *XNArea* argument is dependent on the input method style that has been set.

If the input method style is *XIMPreeditPosition*, *XNArea* specifies the clipping region within which preediting will take place. If the focus window has been set, the coordinates are assumed to be relative to the focus window. Otherwise, the coordinates are assumed to be relative to the client window. If neither has been set, the results are undefined. If *XNArea* is not specified, the input method will default the clipping region to the geometry of the *XNFocusWindow*. If the area specified is NULL or invalid, the results are undefined.

If the input style is *XIMPreeditArea* or *XIMStatusArea*, *XNArea* specifies the geometry provided by the client to the input method. The input method may use this area to display its data, either Preedit or Status depending on the area designated. The input method may create a window as a child of the client window with dimensions that fit the *XNArea*. The coordinates are relative to the client window. If the client window has not been set yet, the input method should save these values and apply them when the client window is set. If *XNArea* is not specified, is set to NULL or is invalid, the results are undefined.

Area Needed

When set, the *XNAreaNeeded* argument specifies the geometry suggested by the client for this area (Preedit or Status). The value associated with the argument must be a pointer to a structure of type *XRectangle*. Note that the x, y values are not used and that non-zero values for width or height are the constraints that the client wishes the input method to respect.

When read, the *XNAreaNeeded* argument specifies the preferred geometry desired by the input method for the area.

This argument is only valid if the input style is *XIMPreeditArea* or *XIMStatusArea*. It is used for geometry negotiation between the client and the input method and has no other effect upon the input method (see Section 14.6.5).

Spot Location

The *XNSpotLocation* argument specifies to the input method the coordinates of the ''spot'' to be used by an input method executing with *XNInputStyle* set to *XIMPreeditPosition*. When specified to any input method other than *XIMPreeditPosition*, this XIC value is ignored.

The x coordinate specifies the position where the next character would be inserted. The y coordinate is the position of the baseline used by current text line in the focus window. The x and y coordinates are relative to the focus window, if it has been set; otherwise, they are relative to the client window. If neither the focus window nor the client window has been set, the results are undefined.

Note that the value of the argument is a pointer to a structure of type *XPoint*.

Colormap

Two different arguments can be used to indicate what colormap the input method should use to allocate colors: a colormap ID, or a standard colormap name.

The *XNColormap* argument is used to specify a colormap ID. The argument value is of type *Colormap*. An invalid argument may generate a *BadColor* error when it is used by the input method.

The *XNStdColormap* argument is used to indicate the name of the standard colormap in which input method should to allocate colors. The argument value is an *Atom* that should be a valid atom for calling *XGetRGBColormaps*. An invalid argument may generate a *BadAtom* error when it is used by the input method.

If colormap is left unspecified, it is defaulted to client window colormap.

Foreground and Background

The *XNForeground* and *XNBackground* arguments specify the foreground and background pixel, respectively. The argument value is of type ''unsigned long''. It must be a valid pixel in the input method colormap.

If these values are left unspecified, the default is determined by the input method.

Background Pixmap

The *XNBackgroundPixmap* argument specifies a background pixmap to be used as the background of the window. The value must be of type *Pixmap*. An invalid argument may generate a *BadPixmap* error when it is used by the input method.

If this value is left unspecified, the default is determined by the input method.

FontSet

The *XNFontSet* argument specifies to the input method what fontset is to be used. The argument value is of type *XFontSet*.

If this value is left unspecified, the default is determined by the input method.

Line Spacing

The *XNLineSpace* argument specifies to the input method what line spacing is to be used in preedit window if more than one line is to be used. This argument is of type ''int''.

If this value is left unspecified, the default is determined by the input method.

Cursor

The *XNCursor* argument specifies to the input method what cursor is to be used in the specified window. This argument is of type *Cursor*.

An invalid argument may generate a *BadCursor* error when it is used by the input method. If this value is left unspecified, the default is determined by the input method.

Preedit and Status Callbacks

A client that wishes to support the input style *XIMPreeditCallbacks* must provide a set of preedit callbacks to the input method. The set of preedit callbacks are as follows:

XNPreeditStartCallback
>   This is called when the input method starts preedit.

XNPreeditDoneCallback
>   This is called when the input method stops preedit.

XNPreeditDrawCallback
>   This is called when a number preedit keystrokes should be echoed.

XNPreeditCaretCallback
>   This is called to move text insertion point within preedit string

A client that wishes to support the input style *XIMStatusCallbacks* must provide a set of status callbacks to the input method. The set of status callbacks are as follows:

XNStatusStartCallback
>   This is called when the input method initializes status area.

XNStatusDoneCallback
>   This is called when the input method no longer needs status area.

XNStatusDrawCallback
>   This is called when updating the status area is required.

The value of any status or preedit argument is a pointer to a structure of type *XIMCallback*.

```
typedef void (*XIMProc)();

typedef struct {
      XPointer client_data;
      XIMProc callback;
} XIMCallback;
```

Each callback has some particular semantics and will carry the data that expresses the environment necessary to the client into a specific data structure. This paragraph only describes the arguments to be used to set the callback. For a complete description of the semantics, see Section 14.11.

Note that setting any of these values while doing preedit may cause unexpected results.

14.11   Callback Semantics

Callbacks are functions defined by clients or text drawing packages that are to be called from the input method when selected events occur. Most clients will use a text editing package or a toolkit and, hence, will not need to define such callbacks. This section defines the callback semantics, when they are triggered, and what their arguments are. Note that this information is mostly useful for X toolkit implementors.

Callbacks are mostly provided so that clients (or text editing packages) can implement on-the-spot preediting in their own window. In that case, the input method needs to communicate and synchronize with the client. Input method needs to communicate changes in the preedit window when it is under control of the client. Those callbacks allow the client to initialize the preedit area, display a new preedit string, move the text insertion point inside preedit, terminate preedit, or update the status area.

All callback functions follow the generic prototype:

```
void CallbackPrototype(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      SomeType call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Specifies data specific to the callback.

The call_data argument is a structure that expresses the arguments needed to achieve the semantics; that is, it is a a specific data structure appropriate to the callback. In cases where no data is needed in the callback, this call_data argument is NULL. The client_data argument is a closure that has been initially specified by the client when specifying the callback and passed back. It may serve, for example, to inherit application context in the callback.

The following paragraphs describe the programming semantics and specific data structure associated with the different reasons.

14.11.1   Geometry Callback

The geometry callback is triggered by the input method to indicate that it wants the client to negotiate geometry. The generic prototype is as follows:

```
void GeometryCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XPointer call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Not used for this callback, always passed as NULL.

Note that a GeometryCallback is called with a NULL call_data argument.

14.11.2 Preedit State Callbacks

When the input method turns input conversion on or off, a PreeditStartCallback or PreeditDoneCallback callback is triggered in order to let the toolkit do the setup or the cleanup for the preedit region.

```
int PreeditStartCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XPointer call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Not used for this callback, always passed as NULL.

When preedit starts on the specified input context, the callback is called with a NULL call_data argument. PreeditStartCallback will return the maximum size of the preedit string. Note that a positive number indicates the maximum number of bytes allowed in the preedit string, and a value of –1 indicates there is no limit.

```
void PreeditDoneCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XPointer call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Not used for this callback, always passed as NULL.

When preedit stops on the specified input context, the callback is called with a NULL call_data argument. The client can release the data allocated by PreeditStartCallback.

PreeditStartCallback should initialize appropriate data needed for displaying preedit information and for handling further PreeditDrawCallback calls. Once PreeditStartCallback is called, it will not be called again before PreeditDoneCallback has been called.

14.11.3 PreeditDraw Callback

This callback is triggered to draw and insert, delete or replace, preedit text in the preedit region. The preedit text may include unconverted input text such as Japanese kana, converted text such as Japanese Kanji characters, or characters of both kinds. That string is either a multibyte or wide character string, whose encoding matches the locale bound to the input context. The callback prototype is as follows:

```
void PreeditDrawCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XIMPreeditDrawCallbackStruct *call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Specifies the preedit drawing information.

The callback is passed a *XIMPreeditDrawCallbackStruct* structure in the call_data argument. The text member of this structure contains the text to be drawn. After the string has been drawn, the

caret should be moved to the specified location.

The *XIMPreeditDrawCallbackStruct* structure is defined as follows:

```
typedef struct _XIMPreeditDrawCallbackStruct {
        int caret;      /* Cursor offset within preedit string */
        int chg_first;  /* Starting change position */
        int chg_length; /* Length of the change in character count */
        XIMText *text;
} XIMPreeditDrawCallbackStruct;
```

The client must keep updating a buffer of the preedit text, the callback arguments referring to indexes in that buffer. The call_data fields have specific meanings according to the operation:

- To indicate text deletion, the call_data specifies a NULL text field. The text to be deleted is then the current text in buffer from position chg_first (starting at zero) on a (character) length of chg_length.

- When text is non-NULL, it indicates insertion or replacement of text in the buffer.

- A positive chg_length indicates that the characters starting from chg_first to ch_first+chg_length must be deleted and must be replaced by text, whose length is specified in the *XIMText* structure.

- A chg_length value of zero indicates that text must be inserted right at the position specified by chg_first. A value of zero for chg_first specifies the first character in the buffer.

- The caret member is an index in the the preedit text buffer that specifies the character after which the cursor should move after text has been drawn or deleted.

```
typedef struct _XIMText {
        unsigned short length;
        XIMFeedback * feedback;
        Bool encoding_is_wchar;
        union {
            char * multi_byte;
            wchar_t * wide_char;
        } string;
} XIMText;
```

The text string passed is actually a structure specifying:

- The length member is the text length in characters.

- The encoding_is_wchar member ia a value that indicates if the text string is encoded in wide character or multibyte format. This value should be set by the client when it sets the callback.

- The string member is the text string.

- The feedback member indicates rendering type.

The feedback member express the types of rendering feedback the callback should apply when drawing text. Rendering of the text to be drawn is specified either in generic ways (for example, primary, secondary) or in specific ways (reverse, underline). When generic indications are given, the client is free to choose the rendering style. It is necessary however that primary and secondary are mapped to two distinct rendering styles.

The feedback member also specifies how the rendering of the text argument should be achieved. If feedback is NULL, then rendering is assumed to be the same as rendering of other characters

in the text entry. Otherwise, it specifies an array defining the rendering of each character of the string (hence the length of the array is length).

If an input method wishes to indicate that it is only updating the feedback of the preedit text without changing the content of it, the *XIMText* structure should contain a NULL value for the string field, the number of characters affected in the length field, and the feedback field should point to an array of *XIMFeedback*.

Each element in the array is a bit mask represented by a value of type *XIMFeedback*. The valid masks names are as follows:

```
typedef unsigned long XIMFeedback;

#define    XIMReverse      1
#define    XIMUnderline    (1L<<1)
#define    XIMHighlight    (1L<<2)
#define    XIMPrimary      (1L<<3)
#define    XIMSecondary    (1L<<4)
#define    XIMTertiary     (1L<<5)
```

14.11.4 PreeditCaretCallback

An input method may have its own ''navigation keys'' to allow the user to move the text insertion point in the preedit area (for example, to move backward or forward). Consequently, input method needs to indicate to the client that it should move the text insertion point. It then calls the PreeditCaretCallback.

```
void PreeditCaretCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XIMPreeditCaretCallbackStruct *call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Specifies the preedit caret information.

The input method will trigger PreeditCaretCallback to move the text insertion point during preedit. The call_data argument contains a pointer to an *XIMPreeditCaretCallbackStruct* structure, which indicates where the caret should be moved. The callback must move the insertion point to its new location and return, in field position, the new offset value from initial position.

The *XIMPreeditCaretCallbackStruct* structure is defined as follows:

```
typedef struct _XIMPreeditCaretCallbackStruct {
      int position;                    /* Caret offset within preedit
                                          string */
      XIMCaretDirection direction; /* Caret moves direction */
      XIMCaretStyle style;         /* Feedback of the caret */
} XIMPreeditCaretCallbackStruct;
```

The *XIMCaretStyle* structure is defined as follows:

```
typedef enum {
      XIMIsInvisible, /* Disable caret feedback */
      XIMIsPrimary,   /* UI defined caret feedback */
      XIMIsSecondary, /* UI defined caret feedback */
} XIMCaretStyle;
```

The *XIMCaretDirection* structure is defined as follows:

```
typedef enum {
      XIMForwardChar, XIMBackwardChar,
      XIMForwardWord, XIMBackwardWord,
      XIMCaretUp, XIMCaretDown,
      XIMNextLine, XIMPreviousLine,
      XIMLineStart, XIMLineEnd,
      XIMAbsolutePosition,
      XIMDontChange,
} XIMCaretDirection;
```

The meaning of these values are defined as follows:

XIMForwardChar
              Move caret forward one character position.

XIMBackwardChar
              Move caret backward one character position.

XIMForwardWord
              Move caret forward one word position.

XIMBackwardWord
              Move caret backward one word position.

XIMCaretUp   Move caret up one line keeping current offset.

XIMCaretDown   Move caret down one line keeping current offset.

XIMPreviousLine   Move caret up one line.

XIMNextLine   Move caret down one line.

XIMLineStart   Move caret to the beginning of the current display line that contains the caret.

XIMLineEnd   Move caret to the end of the current display line that contains the caret.

XIMAbsolutePosition
              The callback must move to the location specified by the position field of the
              callback data, indicated in characters, starting from the beginning of the
              preedit text. Hence, a value of zero means move back to beginning of the
              preedit text.

XIMDontChange   The caret position does not change.

14.11.5 Status Callbacks

An input method may communicate changes in the status of an input context (for example, created, destroyed or focus changes) with three status callbacks: StatusStartCallback, StatusDoneCallback and StatusDrawCallback.

When the input context is created or gains focus, the input method calls the StatusStartCallback callback.

```
void StatusStartCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XPointer call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Not used for this callback, always passed as NULL.

The callback should initialize appropriate data for displaying status and be prepared to further StatusDrawCallback calls. Once StatusStartCallback is called, it will not be called again before StatusDoneCallback has been called.

When an input context is destroyed or when it loses focus, the input method calls StatusDoneCallback.

```
void StatusDoneCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XPointer call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Not used for this callback, always passed as NULL.

The callback may release any data allocated on *StatusStart*.

When an input context status has to be updated, the input method calls *StatusDrawCallback*.

```
void StatusDrawCallback(ic, client_data, call_data)
      XIC ic;
      XPointer client_data;
      XIMStatusDrawCallbackStruct *call_data;
```

ic   Specifies the input context.

client_data   Specifies the additional client data.

call_data   Specifies the status drawing information.

The callback should update the status area by either drawing a string, or imaging a bitmap in the status area.

The *XIMStatusDataType* and *XIMStatusDrawCallbackStruct* structures are defined as follows:

```
typedef enum {
      XIMTextType,
      XIMBitmapType,
} XIMStatusDataType;

typedef struct _XIMStatusDrawCallbackStruct {
      XIMStatusDataType type;
      union {
          XIMText *text;
          Pixmap  bitmap;
      } data;
} XIMStatusDrawCallbackStruct;
```

14.12 Event Filtering

Xlib provides the ability for an input method to register a filter internal to Xlib. This filter is called by a client (or toolkit) by calling *XFilterEvent* after calling *XNextEvent*. Any client that uses the *XIM* interface should call *XFilterEvent* to allow input methods to process their events without knowledge of the client's dispatching mechanism. A client's user interface policy may determine the priority of event filters with respect to other event handling mechanisms (for example, modal grabs).

Clients may not know how many filters there are, if any, and what they do. They may only know if an event has been filtered on return of *XFilterEvent*. Clients should discard filtered events.

```
Bool XFilterEvent(event, w)
      XEvent *event;
      Window w;
```

event   Specifies the event to filter.

w   Specifies the window for which the filter is to be applied.

If the window argument is *None*, *XFilterEvent* applies the filter to the window specified in the *XEvent* structure. The window argument is provided so that layers above Xlib that do event redirection can indicate to which window an event has been redirected.

If *XFilterEvent* returns *True*, then some input method has filtered the event, and the client should discard the event. If *XFilterEvent* returns *False*, then the client should continue processing the event.

If a grab has occurred in the client, and *XFilterEvent* returns *True*, the client should ungrab the keyboard.

                                                      Stamp:XXXXXXXXXXXXXXXXXXXXXXXXX

14.13   Getting Keyboard Input

To get composed input from an input method, use *XmbLookupString* or *XwcLookupString*.

```
int XmbLookupString(ic, event, buffer_return,
        bytes_buffer, keysym_return, status_return)
    XIC ic;
    XKeyPressedEvent *event;
    char *buffer_return;
    int bytes_buffer;
    KeySym *keysym_return;
    Status *status_return;

int XwcLookupString(ic, event, buffer_return,
        bytes_buffer, keysym_return, status_return)
    XIC ic;
    XKeyPressedEvent *event;
    wchar_t *buffer_return;
    int wchars_buffer;
    KeySym *keysym_return;
    Status *status_return;
```

ic	Specifies the input context.
event	Specifies the key event to be used.
buffer_return	Returns a multibyte string or wide character string (if any) from the input method.
bytes_buffer *wchars_buffer*	Specifies space available in return buffer.
keysym_return	Returns the KeySym computed from the event if this argument is not NULL.
status_return	Returns a value indicating what kind of data is returned.

The *XmbLookupString* and *XwcLookupString* functions return the string from the input method specified in the buffer_return argument. If no string is returned, the buffer_return argument is unchanged.

The KeySym into which the KeyCode from the event was mapped is returned in the keysym_return argument if it is non-NULL and the status_return argument indicates that a KeySym was returned. If both a string and a KeySym are returned, the KeySym value does not necessarily correspond to the string returned.

Note that *XmbLookupString* returns the length of the string in bytes and that *XwcLookupString* returns the length of the string in characters.

XmbLookupString and *XwcLookupString* return text in the encoding of the locale bound to the input method of the specified input context.

Note that each string returned by *XmbLookupString* and *XwcLookupString* begins in the initial state of the encoding of the locale (if the encoding of the locale is state-dependent).

Note: In order to insure proper input processing, it is essential that the client pass only *KeyPress* events to *XmbLookupString* and *XwcLookupString*. Their behaviour when a client passes a *KeyRelease* event is undefined.

Clients should check the status_return argument before using the other returned values. These two functions both return a value to status_return that indicates what has been returned in the

other arguments. The possible values returned are:

XBufferOverflow The input string to be returned is too large for the supplied buffer_return. The required size (*XmbLookupString* in bytes; *XwcLookupString* in characters) is returned as the value of the function, and the contents of buffer_return and keysym_return are not modified. The client should recall the function with the same event and a buffer of adequate size in order to obtain the string.

XLookupNone No consistent input has been composed so far. The contents of buffer_return and keysym_return are not modified, and the function returns zero.

XLookupChars Some input characters have been composed. They are placed in the buffer_return argument, and the string length is returned as the value of the function. The string is encoded in the locale bound to the input context. The contents of the keysym_return argument is not modified.

XLookupKeySym
A KeySym has been returned instead of a string and is returned in keysym_return. The contents of the buffer_return argument is not modified, and the function returns zero.

XLookupBoth Both a KeySym and a string are returned; *XLookupChars* and *XLookupKeySym* occur simultaneously.

It does not make any difference if the input context passed as an argument to *XmbLookupString* and *XwcLookupString* is the one currently in possession of the focus or not. Input may have been composed within an input context before it lost the focus, and that input may be returned on subsequent calls to *XmbLookupString* or *XwcLookupString*, even though it does not have any more keyboard focus.

14.14   Input Method Conventions

The input method architecture is transparent to the client. However, clients should respect a number of conventions in order to work properly. Clients must also be aware of possible effects of synchronization between input method and library in the case of a remote input server.

14.14.1 Client Conventions

A well-behaved client (or toolkit) should first query the input method style. If the client cannot satisfy the requirements of the supported styles (in terms of geometry management or callbacks), it should negotiate with the user continuation of the program or raise an exception or error of some sort.

14.14.2 Synchronization Conventions

A *KeyPress* event with a KeyCode of zero is used exclusively as a signal that an input method has composed input which can be returned by *XmbLookupString* or *XwcLookupString*. No other use is made of a *KeyPress* event with KeyCode of zero.

Such an event may be generated by either a front-end or a back-end input method in an implementation-dependent manner. Some possible ways to generate this event include:

- a synthetic event sent by an input method server

- an artificial event created by a input method filter and pushed onto a client's event queue

- a *KeyPress* event whose KeyCode value is modified by an input method filter.

When callback support is specified by client, input methods will not take action unless they explicitly called back the client and obtained no response (the callback is not specified or returned invalid data).

14.15   String Constants

The following symbols for string constants are defined in **<X11/Xlib.h>**. Although they are shown here with particular macro definitions, they may be implemented as macros, as global symbols, or as a mixture of the two. The string pointer value itself is not significant; clients must not assume that inequality of two values implies inequality of the actual string data.

```
#define     XNVaNestedList          "XNVaNestedList"
#define     XNQueryInputStyle       "queryInputStyle"
#define     XNClientWindow          "clientWindow"
#define     XNInputStyle            "inputStyle"
#define     XNFocusWindow           "focusWindow"
#define     XNResourceName          "resourceName"
#define     XNResourceClass         "resourceClass"
#define     XNGeometryCallback      "geometryCallback"
#define     XNFilterEvents          "filterEvents"
#define     XNPreeditStartCallback  "preeditStartCallback"
#define     XNPreeditDoneCallback   "preeditDoneCallback"
#define     XNPreeditDrawCallback   "preeditDrawCallback"
#define     XNPreeditCaretCallback  "preeditCaretCallback"
#define     XNPreeditAttributes     "preeditAttributes"
#define     XNStatusStartCallback   "statusStartCallback"
#define     XNStatusDoneCallback    "statusDoneCallback"
#define     XNStatusDrawCallback    "statusDrawCallback"
#define     XNStatusAttributes      "statusAttributes"
#define     XNArea                  "area"
#define     XNAreaNeeded            "areaNeeded"
#define     XNSpotLocation          "spotLocation"
#define     XNColormap              "colorMap"
#define     XNStdColormap           "stdColorMap"
#define     XNForeground            "foreground"
#define     XNBackground            "background"
#define     XNBackgroundPixmap      "backgroundPixmap"
#define     XNFontSet               "fontSet"
#define     XNLineSpace             "lineSpace"
#define     XNCursor                "cursor"
```

Inter-client Communication Functions

The **ICCCM** specification, hereafter referred to as the ICCCM, details the X Consortium approved conventions that govern inter-client communications. These conventions ensure peer-to-peer client cooperation in the use of selections, cut buffers and shared resources as well as client cooperation with window and session managers. For further information, **ICCCM** specification.

Xlib provides a number of standard properties and programming interfaces that are ICCCM-compliant. The predefined atoms for some of these properties are defined in the **<X11/Xatom.h>** header file, where to avoid name conflicts with user symbols their *#define* name has an XA_ prefix. For further information about atoms and properties, see Section 5.3.

Xlib's selection and cut buffer mechanisms provide the primary programming interfaces by which peer client applications communicate with each other (see Section 5.5 and Section 17.6). The functions discussed in this chapter provide the primary programming interfaces by which client applications communicate with their window and session managers as well as share standard colormaps.

The standard properties that are of special interest for communicating with window and session managers are:

Name	Type	Format	Description
WM_CLASS	STRING	8	Set by application programs to allow window and session managers to obtain the application's resources from the resource database.
WM_CLIENT_MACHINE	TEXT		The string name of the machine on which the client application is running.
WM_COLORMAP_WINDOWS	WINDOW	32	The list of window IDs that may need a different colormap than that of their top-level window.
WM_COMMAND	TEXT		The command and arguments, null-separated, used to invoke the application.
WM_HINTS	WM_HINTS	32	Additional hints set by the client for use by the window manager. The C type of this property is *XWMHints*.
WM_ICON_NAME	TEXT		The name to be used in an icon.
WM_ICON_SIZE	WM_ICON_SIZE	32	The window manager may set this property on the root window to specify the icon sizes it supports. The C type of this property is *XIconSize*.

Name	Type	Format	Description
WM_NAME	TEXT		The name of the application.
WM_NORMAL_HINTS	WM_SIZE_HINTS	32	Size hints for a window in its normal state. The C type of this property is *XSizeHints*.
WM_PROTOCOLS	ATOM	32	List of atoms that identify the communications protocols between the client and window manager in which the client is willing to participate.
WM_STATE	WM_STATE	32	Intended for communication between window and session managers only.
WM_TRANSIENT_FOR	WINDOW	32	Set by application programs to indicate to the window manager that a transient top-level window, such as a dialog box.

The remainder of this chapter discusses:

- client-to-window-manager communication

- client-to-session-manager communication

- standard colormaps.

15.1 Client to Window Manager Communication

This section discusses how to:

- manipulate top-level windows
- convert string lists
- set and read text properties
- set and read the WM_NAME property
- set and read the WM_ICON_NAME property
- set and read the WM_HINTS property
- set and read the WM_NORMAL_HINTS property
- set and read the WM_CLASS property
- set and read the WM_TRANSIENT_FOR property
- set and read the WM_PROTOCOLS property
- set and read the WM_COLORMAP_WINDOWS property
- set and read the WM_ICON_SIZE property
- use window manager convenience functions.

15.1.1 Manipulating Top-level Windows

Xlib provides functions that you can use to change the visibility or size of top-level windows (that is, those that were created as children of the root window). Note that the subwindows that you create are ignored by window managers. Therefore, you should use the basic window functions described in Chapter 4 to manipulate your application's subwindows.

To request that a top-level window be iconified, use *XIconifyWindow*.

```
Status XIconifyWindow(display, w, screen_number)
      Display *display;
      Window w;
      int screen_number;
```

display   Specifies the connection to the X server.

w   Specifies the window.

screen_number   Specifies the appropriate screen number on the host server.

The *XIconifyWindow* function sends a WM_CHANGE_STATE *ClientMessage* event with a format of 32 and a first data element of *IconicState* (as described in Section 5.1 of the **ICCCM** specification) and a window of w to the root window of the specified screen with an event mask set to *SubstructureNotifyMask | SubstructureRedirectMask*. Window managers may elect to receive this message and if the window is in its normal state, may treat it as a request to change the window's state from normal to iconic. If the WM_CHANGE_STATE property cannot be interned, *XIconifyWindow* does not send a message and returns a zero status. It returns a non-zero status if the client message is sent successfully; otherwise, it returns a zero status.

To request that a top-level window be withdrawn, use *XWithdrawWindow*.

```
Status XWithdrawWindow(display, w, screen_number)
      Display *display;
      Window w;
      int screen_number;
```

display   Specifies the connection to the X server.

w   Specifies the window.

screen_number   Specifies the appropriate screen number on the host server.

The *XWithdrawWindow* function unmaps the specified window and sends a synthetic *UnmapNotify* event to the root window of the specified screen. Window managers may elect to receive this message and may treat it as a request to change the window's state to withdrawn. When a window is in the withdrawn state, neither its normal nor its iconic representations is visible. It returns a non-zero status if the *UnmapNotify* event is successfully sent; otherwise, it returns a zero status.

XWithdrawWindow can generate a *BadWindow* error.

To request that a top-level window be reconfigured, use *XReconfigureWMWindow*.

```
Status XReconfigureWMWindow(display, w, screen_number,
      value_mask, values)
      6isplay *display;
      Window w;
      int screen_number;
      unsigned int value_mask;
      XWindowChanges *values;
```

display   Specifies the connection to the X server.

w   Specifies the window.

screen_number   Specifies the appropriate screen number on the host server.

value_mask   Specifies which values are to be set using information in the values structure. This mask is the bitwise inclusive OR of the valid configure window values bits.

values   Specifies the *XWindowChanges* structure.

The *XReconfigureWMWindow* function issues a *ConfigureWindow* request on the specified top-level window. If the stacking mode is changed and the request fails with a *BadMatch* error, the error is trapped by Xlib and a synthetic *ConfigureRequestEvent* containing the same configuration parameters is sent to the root of the specified window. Window managers may elect to receive this event and treat it as a request to reconfigure the indicated window. It returns a non-zero status if the request or event is successfully sent; otherwise, it returns a zero status.

XReconfigureWMWindow can generate *BadValue* and *BadWindow* errors.

15.1.2   Converting String Lists

Many of the text properties allow a variety of types and formats. Because the data stored in these properties are not simple null-terminated strings, a *XTextProperty* structure is used to describe the encoding, type and length of the text as well as its value. The *XTextProperty* structure contains:

```
typedef struct {
      unsigned char *value; /* property data */
      Atom encoding;          /* type of property */
      int format;             /* 8, 16 or 32 */
      unsigned long nitems; /* number of items in value */
} XTextProperty;
```

Xlib provides functions to convert localized text to or from encodings which support the inter-client communication conventions for text. In addition, functions are provided for converting between lists of pointers to character strings and text properties in the STRING encoding.

The functions for localized text return a signed integer error status which encodes *Success* as zero, specific error conditions as negative numbers, and partial conversion as a count of unconvertible characters.

```
#define     XNoMemory              -1
#define     XLocaleNotSupported    -2
#define     XConverterNotFound     -3

typedef enum {
      XStringStyle,        /* STRING */
      XCompoundTextStyle, /* COMPOUND_TEXT */
      XTextStyle,          /* text in owner's encoding (current
                               locale) */
      XStdICCTextStyle    /* STRING, else COMPOUND_TEXT */
} XICCEncodingStyle;
```

To convert a list of text strings to an *XTextProperty* structure, use *XmbTextListToTextProperty* or *XwcTextListToTextProperty*.

```
int XmbTextListToTextProperty(display, list, count,
        style, text_prop_return)
      Display *display;
      char **list;
      int count;
      XICCEncodingStyle style;
      XTextProperty *text_prop_return;


int XwcTextListToTextProperty(display, list, count,
        style, text_prop_return)
      Display *display;
      wchar_t **list;
      int count;
      XICCEncodingStyle style;
      XTextProperty *text_prop_return;
```

display   Specifies the connection to the X server.

list   Specifies a list of null-terminated character strings.

count   Specifies the number of strings specified.

style   Specifies the manner in which the property is encoded.

text_prop_return   Returns the *XTextProperty* structure.

The *XmbTextListToTextProperty* and *XwcTextListToTextProperty* functions set the specified *XTextProperty* value to a set of null-separated elements representing the concatenation of the specified list of null-terminated text strings. A final terminating null is stored at the end of the value field of text_prop_return but is not included in the nitems member.

The functions set the encoding field of text_prop_return to an Atom for the specified display naming the encoding determined by the specified style, and convert the specified text list to this encoding for storage in the text_prop_return value field. If the style *XStringStyle* or *XCompoundTextStyle* is specified, this encoding is ''STRING'' or ''COMPOUND_TEXT'', respectively. If the style *XTextStyle* is specified, this encoding is the encoding of the current locale. If the style *XStdICCTextStyle* is specified, this encoding is ''STRING'' if the text is fully convertible to STRING, else ''COMPOUND_TEXT''.

If insufficient memory is available for the new value string, the functions return *XNoMemory*. If the current locale is not supported, the functions return *XLocaleNotSupported*. In both of these error cases, the functions do not set text_prop_return.

To determine if the functions are guaranteed not to return *XLocaleNotSupported*, use *XSupportsLocale*.

If the supplied text is not fully convertible to the specified encoding, the functions return the number of unconvertible characters. Each unconvertible character is converted to an implementation-defined and encoding-specific default string. Otherwise, the functions return *Success*. Note that full convertibility to all styles except *XStringStyle* is guaranteed.

To free the storage for the value field, use *XFree*.

To obtain a list of text strings from an *XTextProperty* structure, use *XmbTextPropertyToTextList* or *XwcTextPropertyToTextList*.

```
int XmbTextPropertyToTextList(display, text_prop, list_return,
        count_return)
    Display *display;
    XTextProperty *text_prop;
    char ***list_return;
    int *count_return;

int XwcTextPropertyToTextList(display, text_prop, list_return,
        count_return)
    Display *display;
    XTextProperty *text_prop;
    wchar_t ***list_return;
    int *count_return;
```

display   Specifies the connection to the X server.

text_prop   Specifies the *XTextProperty* structure to be used.

list_return   Returns a list of null-terminated character strings.

count_return   Returns the number of strings.

The *XmbTextPropertyToTextList* and *XwcTextPropertyToTextList* functions return a list of text strings in the current locale representing the null-separated elements of the specified *XTextProperty*

structure. The data in text_prop must be format 8.

Multiple elements of the property (for example, the strings in a disjoint text selection) are separated by a null byte. The contents of the property are not required to be null-terminated; any terminating null should not be included in text_prop.nitems.

If insufficient memory is available for the list and its elements, *XmbTextPropertyToTextList* and *XwcTextPropertyToTextList* return *XNoMemory*. If the current locale is not supported, the functions return *XLocaleNotSupported*. Otherwise, if the encoding field of text_prop is not convertible to the encoding of the current locale, the functions return *XConverterNotFound*. For supported locales, existence of a converter from COMPOUND_TEXT, STRING or the encoding of the current locale is guaranteed if *XSupportsLocale* returns *True* for the current locale (but the actual text may contain unconvertible characters). Conversion of other encodings is implementation-dependent. In all of these error cases, the functions do not set any return values.

Otherwise, *XmbTextPropertyToTextList* and *XwcTextPropertyToTextList* return the list of null-terminated text strings to list_return, and the number of text strings to count_return.

If the value field of text_prop is not fully convertible to the encoding of the current locale, the functions return the number of unconvertible characters. Each unconvertible character is converted to a string in the current locale that is specific to the current locale. To obtain the value of this string, use *XDefaultString*. Otherwise, *XmbTextPropertyToTextList* and *XwcTextPropertyToTextList* return *Success*.

To free the storage for the list and its contents returned by *XmbTextPropertyToTextList*, use *XFreeStringList*. To free the storage for the list and its contents returned by *XwcTextPropertyToTextList*, use *XwcFreeStringList*.

To free the in-memory data associated with the specified wide character string list, use *XwcFreeStringList*.

```
void XwcFreeStringList(list)
      wchar_t **list;
```

list   Specifies the list of strings to be freed.

The *XwcFreeStringList* function frees memory allocated by *XwcTextPropertyToTextList*.

To obtain the default string for text conversion in the current locale, use *XDefaultString*.

```
char *XDefaultString()
```

The *XDefaultString* function returns the default string used by Xlib for text conversion (for example, in *XmbTextListToTextProperty*). The default string is the string in the current locale which is output when an unconvertible character is found during text conversion. If the string returned by *XDefaultString* is the empty string (""), no character is output in the converted text. *XDefaultString* does not return NULL.

The string returned by *XDefaultString* is independent of the default string for text drawing; see *XCreateFontSet* to obtain the default string for an *XFontSet*.

The behaviour when an invalid codepoint is supplied to any Xlib function is undefined.

The returned string is null-terminated. It is owned by Xlib and should not be modified or freed by the client. It may be freed after the current locale is changed. Until freed, it will not be modified by Xlib.

To set the specified list of strings in the STRING encoding to a *XTextProperty* structure, use *XStringListToTextProperty*.

```
Status XStringListToTextProperty(list, count, text_prop_return)
        char **list;
        int count;
        XTextProperty *text_prop_return;
```

list   Specifies a list of null-terminated character strings.

count   Specifies the number of strings.

text_prop_return   Returns the *XTextProperty* structure.

The *XStringListToTextProperty* function sets the specified *XTextProperty* to be of type STRING (format 8) with a value representing the concatenation of the specified list of null-separated character strings. An extra null byte (which is not included in the nitems member) is stored at the end of the value field of text_prop_return. The strings are assumed (without verification) to be in the STRING encoding. If insufficient memory is available for the new value string, *XStringListToTextProperty* does not set any fields in the *XTextProperty* structure and returns a zero status. Otherwise, it returns a non-zero status. To free the storage for the value field, use *XFree*.

To obtain a list of strings from a specified *XTextProperty* structure in the STRING encoding, use *XTextPropertyToStringList*.

```
Status XTextPropertyToStringList(text_prop, list_return,
        count_return)
        XTextProperty *text_prop;
        char ***list_return;
        int *count_return;
```

text_prop   Specifies the *XTextProperty* structure to be used.

list_return   Returns a list of null-terminated character strings.

count_return   Returns the number of strings.

The *XTextPropertyToStringList* function returns a list of strings representing the null-separated elements of the specified *XTextProperty* structure. The data in text_prop must be of type STRING and format 8. Multiple elements of the property (for example, the strings in a disjoint text selection) are separated by NULL (encoding 0). The contents of the property are not null-terminated. If insufficient memory is available for the list and its elements, *XTextPropertyToStringList* sets no return values and returns a zero status. Otherwise, it returns a non-zero status. To free the storage for the list and its contents, use *XFreeStringList*.

To free the in-memory data associated with the specified string list, use *XFreeStringList*.

```
void XFreeStringList(list)
        char **list;
```

list   Specifies the list of strings to be freed.

The *XFreeStringList* function releases memory allocated by *XmbTextPropertyToTextList* and *XTextPropertyToStringList*, and the missing charset list allocated by *XCreateFontSet*.

15.1.3   Setting and Reading Text Properties

Xlib provides two functions that you can use to set and read the text properties for a given window. You can use these functions to set and read those properties of type TEXT (WM_NAME, WM_ICON_NAME, WM_COMMAND and WM_CLIENT_MACHINE).   In addition, Xlib provides separate convenience functions that you can use to set each of these properties. For further information about these convenience functions, see Section 15.1.4, Section 15.1.5, Section 15.2.1 and Section 15.2.2, respectively.

To set one of a window's text properties, use *XSetTextProperty*.

```
void XSetTextProperty(display, w, text_prop, property)
      Display *display;
      Window w;
      XTextProperty *text_prop;
      Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop   Specifies the *XTextProperty* structure to be used.

property   Specifies the property name.

The *XSetTextProperty* function replaces the existing specified property for the named window with the data, type, format, and number of items determined by the value field, the encoding field, the format field and the nitems field, respectively, of the specified *XTextProperty* structure. If the property does not already exist, *XSetTextProperty* sets it for the specified window.

XSetTextProperty can generate *BadAlloc*, *BadAtom*, *BadValue* and *BadWindow* errors.

To read one of a window's text properties, use *XGetTextProperty*.

```
Status XGetTextProperty(display, w, text_prop_return, property)
      Display *display;
      Window w;
      XTextProperty *text_prop_return;
      Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop_return   Returns the *XTextProperty* structure.

property   Specifies the property name.

The *XGetTextProperty* function reads the specified property from the window and stores the data in the returned *XTextProperty* structure. It stores the data in the value field, the type of the data in the encoding field, the format of the data in the format field, and the number of items of data in the nitems field. An extra byte containing null (which is not included in the nitems member) is stored at the end of the value field of text_prop_return. The particular interpretation of the property's encoding and data as ''text'' is left to the calling application. If the specified property does not exist on the window, *XGetTextProperty* sets the value field to NULL, the encoding field to None, the format field to zero, and the nitems field to zero.

If it was able to read and store the data in the *XTextProperty* structure, *XGetTextProperty* returns a non-zero status; otherwise, it returns a zero status.

XGetTextProperty can generate *BadAtom* and *BadWindow* errors.

15.1.4 Setting and Reading the WM_NAME Property

Xlib provides convenience functions that you can use to set and read the WM_NAME property for a given window.

To set a window's WM_NAME property with the supplied convenience function, use *XSetWMName.*

```
void XSetWMName(display, w, text_prop)
      Display *display;
      Window w;
      XTextProperty *text_prop;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop   Specifies the *XTextProperty* structure to be used.

The *XSetWMName* convenience function calls *XSetTextProperty* to set the WM_NAME property.

To read a window's WM_NAME property with the supplied convenience function, use *XGetWMName.*

```
Status XGetWMName(display, w, text_prop_return)
      Display *display;
      Window w;
      XTextProperty *text_prop_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop_return Returns the *XTextProperty* structure.

The *XGetWMName* convenience function calls *XGetTextProperty* to obtain the WM_NAME property. It returns non-zero status on success; otherwise it returns a zero status.

The following two functions have been superseded by *XSetWMName* and *XGetWMName*, respectively. You can use these additional convenience functions for window names that are encoded as STRING properties.

To assign a name to a window, use *XStoreName.*

```
XStoreName(display, w, window_name)
      Display *display;
      Window w;
      char *window_name;
```

display   Specifies the connection to the X server.

w   Specifies the window.

window_name   Specifies the window name, which should be a null-terminated string.

The *XStoreName* function assigns the name passed to window_name to the specified window. A window manager can display the window name in some prominent place, such as the title bar, to allow users to identify windows easily. Some window managers may display a window's name in the window's icon, although they are encouraged to use the window's icon name if one is provided by the application. If the string is not in the Host Portable Character Encoding the

result is implementation-dependent.

XStoreName can generate *BadAlloc* and *BadWindow* errors.

To get the name of a window, use *XFetchName*.

```
Status XFetchName(display, w, window_name_return)
      Display *display;
      Window w;
      char **window_name_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

window_name_return
        Returns the window name, which is a null-terminated string.

The *XFetchName* function returns the name of the specified window. If it succeeds, it returns non-zero; otherwise, no name has been set for the window, and it returns zero. If the WM_NAME property has not been set for this window, *XFetchName* sets window_name_return to NULL. If the data returned by the server is in the Latin Portable Character Encoding, then the returned string is in the Host Portable Character Encoding. Otherwise, the result is implementation-dependent. When finished with it, a client must free the window name string using *XFree*.

XFetchName can generate a *BadWindow* error.

15.1.5   Setting and Reading the WM_ICON_NAME Property

Xlib provides convenience functions that you can use to set and read the WM_ICON_NAME property for a given window.

To set a window's WM_ICON_NAME property, use *XSetWMIconName*.

```
void XSetWMIconName(display, w, text_prop)
      Display *display;
      Window w;
      XTextProperty *text_prop;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop   Specifies the *XTextProperty* structure to be used.

The *XSetWMIconName* convenience function calls *XSetTextProperty* to set the WM_ICON_NAME property.

To read a window's WM_ICON_NAME property, use *XGetWMIconName*.

```
Status XGetWMIconName(display, w, text_prop_return)
      Display *display;
      Window w;
      XTextProperty *text_prop_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop_return   Returns the *XTextProperty* structure.

The *XGetWMIconName* convenience function calls *XGetTextProperty* to obtain the WM_ICON_NAME property. It returns non-zero status on success; otherwise it returns a zero status.

The next two functions have been superseded by *XSetWMIconName* and *XGetWMIconName*, respectively. You can use these additional convenience functions for window names that are encoded as STRING properties.

To set the name to be displayed in a window's icon, use *XSetIconName*.

```
XSetIconName(display, w, icon_name)
      Display *display;
      Window w;
      char *icon_name;
```

display   Specifies the connection to the X server.

w   Specifies the window.

icon_name   Specifies the icon name, which should be a null-terminated string.

If the string is not in the Host Portable Character Encoding the result is implementation-dependent. *XSetIconName* can generate *BadAlloc* and *BadWindow* errors.

To get the name a window wants displayed in its icon, use *XGetIconName*.

```
Status XGetIconName(display, w, icon_name_return)
      Display *display;
      Window w;
      char **icon_name_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

icon_name_return Returns the window's icon name, which is a null-terminated string.

The *XGetIconName* function returns the name to be displayed in the specified window's icon. If it succeeds, it returns non-zero; otherwise, if no icon name has been set for the window, it returns zero. If you never assigned a name to the window, *XGetIconName* sets icon_name_return to NULL. If the data returned by the server is in the Latin Portable Character Encoding, then the returned string is in the Host Portable Character Encoding. Otherwise, the result is implementation-dependent. When finished with it, a client must free the icon name string using *XFree*.

XGetIconName can generate a *BadWindow* error.

15.1.6 Setting and Reading the WM_HINTS Property

Xlib provides functions that you can use to set and read the WM_HINTS property for a given window. These functions use the flags and the *XWMHints* structure, as defined in the **<X11/Xutil.h>** header file.

To allocate an *XWMHints* structure, use *XAllocWMHints*.

```
XWMHints *XAllocWMHints()
```

The *XAllocWMHints* function allocates and returns a pointer to a *XWMHints* structure. Note that all fields in the *XWMHints* structure are initially set to zero. If insufficient memory is available, *XAllocWMHints* returns NULL. To free the memory allocated to this structure, use *XFree*.

The *XWMHints* structure contains:

```
/* Window manager hints mask bits */

#define     InputHint           (1L << 0)
#define     StateHint           (1L << 1)
#define     IconPixmapHint      (1L << 2)
#define     IconWindowHint      (1L << 3)
#define     IconPositionHint    (1L << 4)
#define     IconMaskHint        (1L << 5)
#define     WindowGroupHint     (1L << 6)
#define     AllHints            (InputHint|StateHint|IconPixmapHint|
                                    IconWindowHint|IconPositionHint|
                                    IconMaskHint|WindowGroupHint)

/* Values */

typedef struct {
        long flags;         /* marks which fields in this structure are
                                defined */
        Bool input;         /* does this application rely on the window
                                manager to get keyboard input? */
        int initial_state;  /* see below */
        Pixmap icon_pixmap; /* pixmap to be used as icon */
        Window icon_window; /* window to be used as icon */
        int icon_x, icon_y; /* initial position of icon */
        Pixmap icon_mask;   /* pixmap to be used as mask for
                                icon_pixmap */
        XID window_group;   /* id of related window group */
        /* this structure may be extended in the future */
} XWMHints;
```

The input member is used to communicate to the window manager the input focus model used by the application. Applications that expect input but never explicitly set focus to any of their subwindows (that is, use the push model of focus management), such as X Version 10 style applications that use real-estate driven focus, should set this member to *True*. Similarly, applications that set input focus to their subwindows only when it is given to their top-level window by a window manager should also set this member to *True*. Applications that manage their own input focus by explicitly setting focus to one of their subwindows whenever they want keyboard input (that is, use the pull model of focus management) should set this member to *False*. Applications that never expect any keyboard input also should set this member to *False*.

Pull model window managers should make it possible for push model applications to get input by setting input focus to the top-level windows of applications whose input member is *True*. Push model window managers should make sure that pull model applications do not break them by resetting input focus to *PointerRoot* when it is appropriate (for example, whenever an application whose input member is *False* sets input focus to one of its subwindows).

The definitions for the initial_state flag are:

```
#define    WithdrawnState    0
#define    NormalState       1 /* most applications start
                                        this way */
#define    IconicState       3 /* application wants to start
                                        as an icon */
```

The icon_mask specifies which pixels of the icon_pixmap should be used as the icon. This allows for nonrectangular icons. Both icon_pixmap and icon_mask must be bitmaps. The icon_window lets an application provide a window for use as an icon for window managers that support such use. The window_group lets you specify that this window belongs to a group of other windows. For example, if a single application manipulates multiple top-level windows, this allows you to provide enough information that a window manager can iconify all of the windows rather than just the one window.

To set a window's WM_HINTS property, use *XSetWMHints*.

```
XSetWMHints(display, w, wmhints)
      Display *display;
      Window w;
      XWMHints *wmhints;
```

display   Specifies the connection to the X server.

w   Specifies the window.

wmhints   Specifies the *XWMHints* structure to be used.

The *XSetWMHints* function sets the window manager hints that include icon information and location, the initial state of the window, and whether the application relies on the window manager to get keyboard input.

XSetWMHints can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_HINTS property, use *XGetWMHints*.

```
XWMHints *XGetWMHints(display, w)
      Display *display;
      Window w;
```

display   Specifies the connection to the X server.

w   Specifies the window.

The *XGetWMHints* function reads the window manager hints and returns NULL if no WM_HINTS property was set on the window or returns a pointer to a *XWMHints* structure if it succeeds. When finished with the data, free the space used for it by calling *XFree*.

XGetWMHints can generate a *BadWindow* error.

15.1.7 Setting and Reading the WM_NORMAL_HINTS Property

Xlib provides functions that you can use to set or read the WM_NORMAL_HINTS property for a given window. The functions use the flags and the *XSizeHints* structure, as defined in the **<X11/Xutil.h>** header file.

To allocate an *XSizeHints* structure, use *XAllocSizeHints*.

```
XSizeHints *XAllocSizeHints()
```

The *XAllocSizeHints* function allocates and returns a pointer to a *XSizeHints* structure. Note that all fields in the *XSizeHints* structure are initially set to zero. If insufficient memory is available,

XAllocSizeHints returns NULL. To free the memory allocated to this structure, use *XFree*.

The *XSizeHints* structure contains:

```
/* Size hints mask bits */

#define      USPosition      (1L << 0) /* user specified x, y */
#define      USSize          (1L << 1) /* user specified width, height */
#define      PPosition       (1L << 2) /* program specified position */
#define      PSize           (1L << 3) /* program specified size */
#define      PMinSize        (1L << 4) /* program specified minimum size */
#define      PMaxSize        (1L << 5) /* program specified maximum size */
#define      PResizeInc      (1L << 6) /* program specified resize
                                               increments */
#define      PAspect         (1L << 7) /* program specified min and
                                               max aspect ratios */
#define      PBaseSize       (1L << 8)
#define      PWinGravity     (1L << 9)
#define      PAllHints       (PPosition|PSize|PMinSize|
                                     PMaxSize|PResizeInc|PAspect)

/* Values */

typedef struct {
      long flags;                       /* marks which fields in this
                                           structure are defined */
      int x, y;                         /* Obsolete */
      int width, height;                /* Obsolete */
      int min_width, min_height;
      int max_width, max_height;
      int width_inc, height_inc;
      struct {
            int x;                       /* numerator */
            int y;                       /* denominator */
      } min_aspect, max_aspect;
      int base_width, base_height;
      int win_gravity;
} XSizeHints;
```

The x, y, width and height members are now obsolete and are left solely for compatibility reasons. The min_width and min_height members specify the minimum window size that still allows the application to be useful. The max_width and max_height members specify the maximum window size. The width_inc and height_inc members define an arithmetic progression of sizes (minimum to maximum) into which the window prefers to be resized. The min_aspect and max_aspect members are expressed as ratios of x and y, and they allow an application to specify the range of aspect ratios it prefers. The base_width and base_height members define the desired size of the window. The window manager will interpret the position of the window and its border width to position the point of the outer rectangle of the overall window specified by the win_gravity member. The outer rectangle of the window includes any borders or decorations supplied by the window manager. In other words, if the window manager decides to place the window where the client asked, the position on the parent window's border named by the win_gravity will be placed where the client window would have been placed in the absence of a window manager.

Note that use of the *PAllHints* macro is highly discouraged.

To set a window's WM_NORMAL_HINTS property, use *XSetWMNormalHints*.

```
void XSetWMNormalHints(display, w, hints)
      Display *display;
      Window w;
      XSizeHints *hints;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints   Specifies the size hints for the window in its normal state.

The *XSetWMNormalHints* function replaces the size hints for the WM_NORMAL_HINTS property on the specified window. If the property does not already exist, *XSetWMNormalHints* sets the size hints for the WM_NORMAL_HINTS property on the specified window. The property is stored with a type of WM_SIZE_HINTS and a format of 32.

XSetWMNormalHints can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_NORMAL_HINTS property, use *XGetWMNormalHints*.

```
Status XGetWMNormalHints(display, w, hints_return,
          supplied_return)
      Display *display;
      Window w;
      XSizeHints *hints_return;
      long *supplied_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints_return   Returns the size hints for the window in its normal state.

supplied_return Returns the hints that were supplied by the user.

The *XGetWMNormalHints* function returns the size hints stored in the WM_NORMAL_HINTS property on the specified window. If the property is of type WM_SIZE_HINTS, is of format 32, and is long enough to contain either an old (pre-ICCCM) or new size hints structure, *XGetWMNormalHints* sets the various fields of the *XSizeHints* structure, sets the supplied_return argument to the list of fields that were supplied by the user (whether or not they contained defined values), and returns a non-zero status. Otherwise, it returns a zero status.

If *XGetWMNormalHints* returns successfully and a pre-ICCCM size hints property is read, the supplied_return argument will contain the following bits:

```
(USPosition|USSize|PPosition|PSize|PMinSize|
 PMaxSize|PResizeInc|PAspect)
```

If the property is large enough to contain the base size and window gravity fields as well, the supplied_return argument will also contain the following bits:

```
PBaseSize|PWinGravity
```

XGetWMNormalHints can generate a *BadWindow* error.

To set a window's WM_SIZE_HINTS property, use *XSetWMSizeHints*.

```
    void XSetWMSizeHints(display, w, hints, property)
        Display *display;
        Window w;
        XSizeHints *hints;
        Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints   Specifies the *XSizeHints* structure to be used.

property   Specifies the property name.

The *XSetWMSizeHints* function replaces the size hints for the specified property on the named window. If the specified property does not already exist, *XSetWMSizeHints* sets the size hints for the specified property on the named window. The property is stored with a type of WM_SIZE_HINTS and a format of 32. To set a window's normal size hints, you can use the *XSetWMNormalHints* function.

XSetWMSizeHints can generate *BadAlloc*, *BadAtom* and *BadWindow* errors.

To read a window's WM_SIZE_HINTS property, use *XGetWMSizeHints*.

```
    Status XGetWMSizeHints(display, w, hints_return,
            supplied_return, property)
        Display *display;
        Window w;
        XSizeHints *hints_return;
        long *supplied_return;
        Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints_return   Returns the *XSizeHints* structure.

supplied_return   Returns the hints that were supplied by the user.

property   Specifies the property name.

The *XGetWMSizeHints* function returns the size hints stored in the specified property on the named window. If the property is of type WM_SIZE_HINTS, is of format 32, and is long enough to contain either an old (pre-ICCCM) or new size hints structure, *XGetWMSizeHints* sets the various fields of the *XSizeHints* structure, sets the supplied_return argument to the list of fields that were supplied by the user (whether or not they contained defined values), and returns a non-zero status. Otherwise, it returns a zero status. To get a window's normal size hints, you can use the *XGetWMNormalHints* function.

If *XGetWMSizeHints* returns successfully and a pre-ICCCM size hints property is read, the supplied_return argument will contain the following bits:

```
    (USPosition|USSize|PPosition|PSize|PMinSize|
     PMaxSize|PResizeInc|PAspect)
```

If the property is large enough to contain the base size and window gravity fields as well, the supplied_return argument will also contain the following bits:

```
    PBaseSize|PWinGravity
```

XGetWMSizeHints can generate *BadAtom* and *BadWindow* errors.

15.1.8   Setting and Reading the WM_CLASS Property

Xlib provides functions that you can use to set and get the WM_CLASS property for a given
window. These functions use the *XClassHint* structure, which is defined in the **<X11/Xutil.h>**
header file.

To allocate an *XClassHint* structure, use *XAllocClassHint*.

```
XClassHint *XAllocClassHint( )
```

The *XAllocClassHint* function allocates and returns a pointer to a *XClassHint* structure. Note that
the pointer fields in the *XClassHint* structure are initially set to NULL. If insufficient memory is
available, *XAllocClassHint* returns NULL. To free the memory allocated to this structure, use
XFree.

The *XClassHint* contains:

```
typedef struct {
        char *res_name;
        char *res_class;
} XClassHint;
```

The res_name member contains the application name, and the res_class member contains the
application class. Note that the name set in this property may differ from the name set as
WM_NAME. That is, WM_NAME specifies what should be displayed in the title bar and,
therefore, can contain temporal information (for example, the name of a file currently in an
editor's buffer). On the other hand, the name specified as part of WM_CLASS is the formal name
of the application that should be used when retrieving the application's resources from the
resource database.

To set a window's WM_CLASS property, use *XSetClassHint*.

```
XSetClassHint(display, w, class_hints)
        Display *display;
        Window w;
        XClassHint *class_hints;
```

display   Specifies the connection to the X server.

w   Specifies the window.

class_hints   Specifies the *XClassHint* structure that is to be used.

The *XSetClassHint* function sets the class hint for the specified window. If the strings are not in
the Host Portable Character Encoding the result is implementation-dependent.

XSetClassHint can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_CLASS property, use *XGetClassHint*.

```
Status XGetClassHint(display, w, class_hints_return)
        Display *display;
        Window w;
        XClassHint *class_hints_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

class_hints_return Returns the *XClassHint* structure.

The *XGetClassHint* function returns the class hint of the specified window to the members of the supplied structure. If the data returned by the server is in the Latin Portable Character Encoding, then the returned strings are in the Host Portable Character Encoding. Otherwise, the result is implementation-dependent. It returns non-zero status on success; otherwise it returns a zero status. To free res_name and res_class when finished with the strings, use *XFree* on each individually.

XGetClassHint can generate a *BadWindow* error.

15.1.9   Setting and Reading the WM_TRANSIENT_FOR Property

Xlib provides functions that you can use to set and read the WM_TRANSIENT_FOR property for a given window.

To set a window's WM_TRANSIENT_FOR property, use *XSetTransientForHint*.

```
XSetTransientForHint(display, w, prop_window)
      Display *display;
      Window w;
      Window prop_window;
```

display   Specifies the connection to the X server.

w   Specifies the window.

prop_window   Specifies the window that the WM_TRANSIENT_FOR property is to be set to.

The *XSetTransientForHint* function sets the WM_TRANSIENT_FOR property of the specified window to the specified prop_window.

XSetTransientForHint can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_TRANSIENT_FOR property, use *XGetTransientForHint*.

```
Status XGetTransientForHint(display, w, prop_window_return)
      Display *display;
      Window w;
      Window *prop_window_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

prop_window_return Returns the WM_TRANSIENT_FOR property of the specified window.

The *XGetTransientForHint* function returns the WM_TRANSIENT_FOR property for the specified window. It returns non-zero status on success; otherwise it returns a zero status.

XGetTransientForHint can generate a *BadWindow* error.

15.1.10 Setting and Reading the WM_PROTOCOLS Property

Xlib provides functions that you can use to set and read the WM_PROTOCOLS property for a given window.

To set a window's WM_PROTOCOLS property, use *XSetWMProtocols*.

```
Status XSetWMProtocols(display, w, protocols, count)
      Display *display;
      Window w;
      Atom *protocols;
      int count;
```

display   Specifies the connection to the X server.

w   Specifies the window.

protocols   Specifies the list of protocols.

count   Specifies the number of protocols in the list.

The *XSetWMProtocols* function replaces the WM_PROTOCOLS property on the specified window with the list of atoms specified by the protocols argument. If the property does not already exist, *XSetWMProtocols* sets the WM_PROTOCOLS property on the specified window to the list of atoms specified by the protocols argument. The property is stored with a type of ATOM and a format of 32. If it cannot intern the WM_PROTOCOLS atom, *XSetWMProtocols* returns a zero status. Otherwise, it returns a non-zero status.

XSetWMProtocols can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_PROTOCOLS property, use *XGetWMProtocols*.

```
Status XGetWMProtocols(display, w, protocols_return,
          count_return)
      Display *display;
      Window w;
      Atom **protocols_return;
      int *count_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

protocols_return Returns the list of protocols.

count_return   Returns the number of protocols in the list.

The *XGetWMProtocols* function returns the list of atoms stored in the WM_PROTOCOLS property on the specified window. These atoms describe window manager protocols in which the owner of this window is willing to participate. If the property exists, is of type ATOM, is of format 32, and the atom WM_PROTOCOLS can be interned, *XGetWMProtocols* sets the protocols_return argument to a list of atoms, sets the count_return argument to the number of elements in the list, and returns a non-zero status. Otherwise, it sets neither of the return arguments and returns a zero status. To release the list of atoms, use *XFree*.

XGetWMProtocols can generate a *BadWindow* error.

15.1.11 Setting and Reading the WM_COLORMAP_WINDOWS Property

Xlib provides functions that you can use to set and read the WM_COLORMAP_WINDOWS property for a given window.

To set a window's WM_COLORMAP_WINDOWS property, use *XSetWMColormapWindows*.

```
Status XSetWMColormapWindows(display, w, colormap_windows,
        count)
     Display *display;
     Window w;
     Window *colormap_windows;
     int count;
```

display   Specifies the connection to the X server.

w   Specifies the window.

colormap_windows
                   Specifies the list of windows.

count   Specifies the number of windows in the list.

The *XSetWMColormapWindows* function replaces the WM_COLORMAP_WINDOWS property on the specified window with the list of windows specified by the colormap_windows argument. It the property does not already exist, *XSetWMColormapWindows* sets the WM_COLORMAP_WINDOWS property on the specified window to the list of windows specified by the colormap_windows argument. The property is stored with a type of WINDOW and a format of 32. If it cannot intern the WM_COLORMAP_WINDOWS atom, *XSetWMColormapWindows* returns a zero status. Otherwise, it returns a non-zero status.

XSetWMColormapWindows can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_COLORMAP_WINDOWS property, use *XGetWMColormapWindows*.

```
Status XGetWMColormapWindows(display, w, colormap_windows_return,
        count_return)
     Display *display;
     Window w;
     Window **colormap_windows_return;
     int *count_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

colormap_windows_return
                   Returns the list of windows.

count_return   Returns the number of windows in the list.

The *XGetWMColormapWindows* function returns the list of window identifiers stored in the WM_COLORMAP_WINDOWS property on the specified window. These identifiers indicate the colormaps that the window manager may need to install for this window. If the property exists, is of type WINDOW, is of format 32, and the atom WM_COLORMAP_WINDOWS can be interned, *XGetWMColormapWindows* sets the windows_return argument to a list of window identifiers, sets the count_return argument to the number of elements in the list, and returns a non-zero status. Otherwise, it sets neither of the return arguments and returns a zero status. To release the list of window identifiers, use *XFree*.

XGetWMColormapWindows can generate a *BadWindow* error.

15.1.12 **Setting and Reading the WM_ICON_SIZE Property**

Xlib provides functions that you can use to set and read the WM_ICON_SIZE property for a given window. These functions use the *XIconSize* structure, which is defined in the <**X11/Xutil.h**> header file.

To allocate an *XIconSize* structure, use *XAllocIconSize*.

```
XIconSize *XAllocIconSize( )
```

The *XAllocIconSize* function allocates and returns a pointer to a *XIconSize* structure. Note that all fields in the *XIconSize* structure are initially set to zero. If insufficient memory is available, *XAllocIconSize* returns NULL. To free the memory allocated to this structure, use *XFree*.

The *XIconSize* structure contains:

```
typedef struct {
        int min_width, min_height;
        int max_width, max_height;
        int width_inc, height_inc;
} XIconSize;
```

The width_inc and height_inc members define an arithmetic progression of sizes (minimum to maximum) that represent the supported icon sizes.

To set a window's WM_ICON_SIZE property, use *XSetIconSizes*.

```
XSetIconSizes(display, w, size_list, count)
        Display *display;
        Window w;
        XIconSize *size_list;
        int count;
```

display   Specifies the connection to the X server.

w   Specifies the window.

size_list   Specifies the size list.

count   Specifies the number of items in the size list.

The *XSetIconSizes* function is used only by window managers to set the supported icon sizes.

XSetIconSizes can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_ICON_SIZE property, use *XGetIconSizes*.

```
Status XGetIconSizes(display, w, size_list_return, count_return)
        Display *display;
        Window w;
        XIconSize **size_list_return;
        int *count_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

size_list_return   Returns the size list.

count_return   Returns the number of items in the size list.

The *XGetIconSizes* function returns zero if a window manager has not set icon sizes; otherwise, it return non-zero. *XGetIconSizes* should be called by an application that wants to find out what icon sizes would be most appreciated by the window manager under which the application is running. The application should then use *XSetWMHints* to supply the window manager with an icon pixmap or window in one of the supported sizes. To free the data allocated in size_list_return, use *XFree*.

XGetIconSizes can generate a *BadWindow* error.

15.1.13   Using Window Manager Convenience Functions

The *XmbSetWMProperties* function stores the standard set of window manager properties, with text properties in standard encodings for internationalized text communication. The standard window manager properties for a given window are WM_NAME, WM_ICON_NAME, WM_HINTS, WM_NORMAL_HINTS, WM_CLASS, WM_COMMAND, WM_CLIENT_MACHINE and WM_LOCALE_NAME.

```
void XmbSetWMProperties(display, w, window_name, icon_name,
        argv, argc, normal_hints, wm_hints, class_hints)
    Display *display;
    Window w;
    char *window_name;
    char *icon_name;
    char *argv[];
    int argc;
    XSizeHints *normal_hints;
    XWMHints *wm_hints;
    XClassHint *class_hints;
```

display	Specifies the connection to the X server.
w	Specifies the window.
window_name	Specifies the window name, which should be a null-terminated string.
icon_name	Specifies the icon name, which should be a null-terminated string.
argv	Specifies the application's argument list.
argc	Specifies the number of arguments.
hints	Specifies the size hints for the window in its normal state.
wm_hints	Specifies the *XWMHints* structure to be used.
class_hints	Specifies the *XClassHint* structure to be used.

The *XmbSetWMProperties* convenience function provides a simple programming interface for setting those essential window properties that are used for communicating with other clients (particularly window and session managers).

If the window_name argument is non-NULL, *XmbSetWMProperties* sets the WM_NAME property. If the icon_name argument is non-NULL, *XmbSetWMProperties* sets the WM_ICON_NAME property. The window_name and icon_name arguments are null-terminated strings in the encoding of the current locale. If the arguments can be fully converted to the STRING encoding, the properties are created with type ''STRING'': otherwise, the arguments are converted to Compound Text, and the properties are created with type ''COMPOUND_TEXT''.

If the normal_hints argument is non-NULL, *XmbSetWMProperties* calls *XSetWMNormalHints*, which sets the WM_NORMAL_HINTS property (see Section 15.1.7). If the wm_hints argument is non-NULL, *XmbSetWMProperties* calls *XSetWMHints*, which sets the WM_HINTS property (see Section 15.1.6).

If the argv argument is non-NULL, *XmbSetWMProperties* sets the WM_COMMAND property from argv and argc. Note that an argc of 0 indicates a zero-length command.

The hostname of this machine is stored using *XSetWMClientMachine* (see Section 15.2.2).

If the class_hints argument is non-NULL, *XmbSetWMProperties* sets the WM_CLASS property. If the res_name member in the *XClassHint* structure is set to the NULL pointer and the RESOURCE_NAME environment variable is set, the value of the environment variable is substituted for res_name. If the res_name member is NULL, the environment variable is not set, and argv and argv[0] are set, then the value of argv[0], stripped of any directory prefixes, is substituted for res_name.

It is assumed that the supplied class_hints.res_name and argv, the RESOURCE_NAME environment variable, and the hostname of this machine are in the encoding of the locale announced for the LC_CTYPE category. (On POSIX-compliant systems, the LC_CTYPE, else LANG environment variable). The corresponding WM_CLASS, WM_COMMAND, and WM_CLIENT_MACHINE properties are typed according to the local host locale announcer. No encoding conversion is performed prior to storage in the properties.

For clients that need to process the property text in a locale, *XmbSetWMProperties* sets the WM_LOCALE_NAME property to be the name of the current locale. The name is assumed to be in the Host Portable Character Encoding, and is converted to STRING for storage in the property.

XmbSetWMProperties can generate *BadAlloc* and *BadWindow* errors.

To set a window's standard window manager properties with strings in the STRING encoding, use *XSetWMProperties*. The standard window manager properties for a given window are WM_NAME, WM_ICON_NAME, WM_HINTS, WM_NORMAL_HINTS, WM_CLASS, WM_COMMAND and WM_CLIENT_MACHINE.

```
void XSetWMProperties(display, w, window_name, icon_name,
        argv, argc, normal_hints, wm_hints, class_hints)
    Display *display;
    Window w;
    XTextProperty *window_name;
    XTextProperty *icon_name;
    char **argv;
    int argc;
    XSizeHints *normal_hints;
    XWMHints *wm_hints;
    XClassHint *class_hints;
```

display   Specifies the connection to the X server.

w   Specifies the window.

window_name   Specifies the window name, which should be a null-terminated string.

icon_name   Specifies the icon name, which should be a null-terminated string.

argv   Specifies the application's argument list.

argc   Specifies the number of arguments.

normal_hints   Specifies the size hints for the window in its normal state.

wm_hints   Specifies the *XWMHints* structure to be used.

class_hints   Specifies the *XClassHint* structure to be used.

The *XSetWMProperties* convenience function provides a single programming interface for setting those essential window properties that are used for communicating with other clients (particularly window and session managers).

If the window_name argument is non-NULL, *XSetWMProperties* calls *XSetWMName*, which in turn, sets the WM_NAME property (see Section 15.1.4). If the icon_name argument is non-NULL, *XSetWMProperties* calls *XSetWMIconName*, which sets the WM_ICON_NAME property (see Section 15.1.5). If the argv argument is non-NULL, *XSetWMProperties* calls *XSetCommand*, which sets the WM_COMMAND property (see Section 15.2.1). Note that an argc of zero is allowed to indicate a zero-length command. Note also that the hostname of this machine is stored using *XSetWMClientMachine* (see Section 15.2.2).

If the normal_hints argument is non-NULL, *XSetWMProperties* calls *XSetWMNormalHints*, which sets the WM_NORMAL_HINTS property (see Section 15.1.7). If the wm_hints argument is non-NULL, *XSetWMProperties* calls *XSetWMHints*, which sets the WM_HINTS property (see Section 15.1.6).

If the class_hints argument is non-NULL, *XSetWMProperties* calls *XSetClassHint*, which sets the WM_CLASS property (see Section 15.1.8). If the res_name member in the *XClassHint* structure is set to the NULL pointer and the RESOURCE_NAME environment variable is set, then the value of the environment variable is substituted for res_name. If the res_name member is NULL, the environment variable is not set, and argv and argv[0] are set, then the value of argv[0], stripped of any directory prefixes, is substituted for res_name.

XSetWMProperties can generate *BadAlloc* and *BadWindow* errors.

15.2   Client to Session Manager Communication

This section discusses how to:

- set and read the WM_COMMAND property

- set and read the WM_CLIENT_MACHINE property.

15.2.1   Setting and Reading the WM_COMMAND Property

Xlib provides functions that you can use to set and read the WM_COMMAND property for a
given window.

To set a window's WM_COMMAND property, use *XSetCommand*.

```
XSetCommand(display, w, argv, argc)
        Display *display;
        Window w;
        char **argv;
        int argc;
```

display   Specifies the connection to the X server.

w   Specifies the window.

argv   Specifies the application's argument list.

argc   Specifies the number of arguments.

The *XSetCommand* function sets the command and arguments used to invoke the application.
(Typically, argv is the argv array of your main program.) If the strings are not in the Host
Portable Character Encoding the result is implementation-dependent.

XSetCommand can generate *BadAlloc* and *BadWindow* errors.

To read a window's WM_COMMAND property, use *XGetCommand*.

```
Status XGetCommand(display, w, argv_return, argc_return)
        Display *display;
        Window w;
        char ***argv_return;
        int *argc_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

argv_return   Returns the application's argument list.

argc_return   Returns the number of arguments returned.

The *XGetCommand* function reads the WM_COMMAND property from the specified window
and returns a string list. If the WM_COMMAND property exists, it is of type STRING and
format 8. If sufficient memory can be allocated to contain the string list, *XGetCommand* fills in
the argv_return and argc_return arguments and returns a non-zero status. Otherwise, it returns
a zero status. If the data returned by the server is in the Latin Portable Character Encoding, then
the returned strings are in the Host Portable Character Encoding. Otherwise, the result is
implementation-dependent. To free the memory allocated to the string list, use *XFreeStringList*.

15.2.2   Setting and Reading the WM_CLIENT_MACHINE Property

Xlib provides functions that you can use to set and read the WM_CLIENT_MACHINE property for a given window.

To set a window's WM_CLIENT_MACHINE property, use *XSetWMClientMachine*.

```
void XSetWMClientMachine(display, w, text_prop)
      Display *display;
      Window w;
      XTextProperty *text_prop;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop   Specifies the *XTextProperty* structure to be used.

The *XSetWMClientMachine* convenience function calls *XSetTextProperty* to set the WM_CLIENT_MACHINE property.

To read a window's WM_CLIENT_MACHINE property, use *XGetWMClientMachine*.

```
Status XGetWMClientMachine(display, w, text_prop_return)
      Display *display;
      Window w;
      XTextProperty *text_prop_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

text_prop_return Returns the *XTextProperty* structure.

The *XGetWMClientMachine* convenience function performs an *XGetTextProperty* on the WM_CLIENT_MACHINE property. It returns non-zero status on success; otherwise it returns a zero status.

15.3   Standard Colormaps

Applications with color palettes, smooth-shaded drawings or digitized images demand large numbers of colors. In addition, these applications often require an efficient mapping from color triples to pixel values that display the appropriate colors.

As an example, consider a three-dimensional display program that wants to draw a smoothly shaded sphere. At each pixel in the image of the sphere, the program computes the intensity and color of light reflected back to the viewer. The result of each computation is a triple of RGB coefficients in the range 0.0 to 1.0. To draw the sphere, the program needs a colormap that provides a large range of uniformly distributed colors. The colormap should be arranged so that the program can convert its RGB triples into pixel values very quickly, because drawing the entire sphere requires many such conversions.

On many current workstations, the display is limited to 256 or fewer colors. Applications must allocate colors carefully, not only to make sure they cover the entire range they need but also to make use of as many of the available colors as possible. On a typical X display, many applications are active at once. Most workstations have only one hardware look-up table for colors, so only one application colormap can be installed at a given time. The application using the installed colormap is displayed correctly, and the other applications ''go technicolor'' and are displayed with false colors.

As another example, consider a user who is running an image processing program to display earth-resources data. The image processing program needs a colormap set up with 8 reds, 8 greens and 4 blues, for a total of 256 colors. Because some colors are already in use in the default colormap, the image processing program allocates and installs a new colormap.

The user decides to alter some of the colors in the image by invoking a color palette program to mix and choose colors. The color palette program also needs a colormap with eight reds, eight greens and four blues, so just like the image processing program, it must allocate and install a new colormap.

Because only one colormap can be installed at a time, the color palette may be displayed incorrectly whenever the image processing program is active. Conversely, whenever the palette program is active, the image may be displayed incorrectly. The user can never match or compare colors in the palette and image. Contention for colormap resources can be reduced if applications with similar color needs share colormaps.

The image processing program and the color palette program could share the same colormap if there existed a convention that described how the colormap was set up. Whenever either program was active, both would be displayed correctly.

The standard colormap properties define a set of commonly used colormaps. Applications that share these colormaps and conventions display true colors more often and provide a better interface to the user.

Standard colormaps allow applications to share commonly used color resources. This allows many applications to be displayed in true colors simultaneously, even when each application needs an entirely filled colormap.

Several standard colormaps are described in this section. Usually, a window manager creates these colormaps. Applications should use the standard colormaps if they already exist.

To allocate an *XStandardColormap* structure, use *XAllocStandardColormap*.

```
XStandardColormap *XAllocStandardColormap()
```

The *XAllocStandardColormap* function allocates and returns a pointer to a *XStandardColormap* structure. Note that all fields in the *XStandardColormap* structure are initially set to zero. If

insufficient memory is available, *XAllocStandardColormap* returns NULL. To free the memory allocated to this structure, use *XFree*.

The *XStandardColormap* structure contains:

```
/* Hints */

#define    ReleaseByFreeingColormap    ( (XID) 1L)

/* Values */

typedef struct {
       Colormap colormap;
       unsigned long red_max;
       unsigned long red_mult;
       unsigned long green_max;
       unsigned long green_mult;
       unsigned long blue_max;
       unsigned long blue_mult;
       unsigned long base_pixel;
       VisualID visualid;
       XID killid;
} XStandardColormap;
```

The colormap member is the colormap created by the *XCreateColormap* function. The red_max, green_max and blue_max members give the maximum red, green and blue values, respectively. Each color coefficient ranges from zero to its max, inclusive. For example, a common colormap allocation is 3/3/2 (3 planes for red, 3 planes for green and 2 planes for blue). This colormap would have red_max = 7, green_max = 7 and blue_max = 3. An alternate allocation that uses only 216 colors is red_max = 5, green_max = 5 and blue_max = 5.

The red_mult, green_mult and blue_mult members give the scale factors used to compose a full pixel value. (See the discussion of the base_pixel members for further information.) For a 3/3/2 allocation, red_mult might be 32, green_mult might be 4, and blue_mult might be 1. For a 6-colors-each allocation, red_mult might be 36, green_mult might be 6, and blue_mult might be 1.

The base_pixel member gives the base pixel value used to compose a full pixel value. Usually, the base_pixel is obtained from a call to the *XAllocColorPlanes* function. Given integer red, green and blue coefficients in their appropriate ranges, one then can compute a corresponding pixel value by using the following expression:

```
(r * red_mult + g * green_mult + b * blue_mult + base_pixel) & 0xFFFFFFFF
```

For *GrayScale* colormaps, only the colormap, red_max, red_mult and base_pixel members are defined. The other members are ignored. To compute a *GrayScale* pixel value, use the following expression:

```
(gray * red_mult + base_pixel) & 0xFFFFFFFF
```

Negative multipliers can be represented by converting the 2's complement representation of the multiplier into an unsigned long and storing the result in the appropriate _mult field. The step of masking by 0xFFFFFFFF effectively converts the resulting positive multiplier into a negative one. The masking step will take place automatically on many machine architectures, depending on the size of the integer type used to do the computation,

The visualid member gives the ID number of the visual from which the colormap was created. The killid member gives a resource ID that indicates whether the cells held by this standard colormap are to be released by freeing the colormap ID or by calling the *XKillClient* function on

the indicated resource. (Note that this method is necessary for allocating out of an existing colormap.)

The properties containing the *XStandardColormap* information have the type RGB_COLOR_MAP.

The remainder of this section discusses standard colormap properties and atoms as well as how to manipulate standard colormaps.

15.3.1   Standard Colormap Properties and Atoms

Several standard colormaps are available. Each standard colormap is defined by a property, and each such property is identified by an atom. The following list names the atoms and describes the colormap associated with each one. The **<X11/Xatom.h>** header file contains the definitions for each of the following atoms, which are prefixed with XA_.

RGB_DEFAULT_MAP

This atom names a property. The value of the property is an array of *XStandardColormap* structures. Each entry in the array describes an RGB subset of the default color map for the Visual specified by visual_id.

Some applications only need a few RGB colors and may be able to allocate them from the system default colormap. This is the ideal situation because the fewer colormaps that are active in the system the more applications are displayed with correct colors at all times.

A typical allocation for the RGB_DEFAULT_MAP on 8-plane displays is 6 reds, 6 greens and 6 blues. This gives 216 uniformly distributed colors (6 intensities of 36 different hues) and still leaves 40 elements of a 256-element colormap available for special-purpose colors for text, borders, and so on.

RGB_BEST_MAP

This atom names a property. The value of the property is an *XStandardColormap*.

The property defines the best RGB colormap available on the screen. (Of course, this is a subjective evaluation.) Many image processing and three-dimensional applications need to use all available colormap cells and to distribute as many perceptually distinct colors as possible over those cells. This implies that there may be more green values available than red, as well as more green or red than blue.

For an 8-plane *PseudoColor* visual, RGB_BEST_MAP is likely to be a 3/3/2 allocation. For a 24-plane *DirectColor* visual, RGB_BEST_MAP is normally an 8/8/8 allocation.

RGB_RED_MAP
RGB_GREEN_MAP
RGB_BLUE_MAP

These atoms name properties. The value of each property is an *XStandardColormap*.

The properties define all-red, all-green and all-blue colormaps, respectively. These maps are used by applications that want to make color-separated images. For example, a user might generate a full-color image on an 8-plane display both by rendering an image three times (once with high color resolution in red, once with green, and once with blue) and by multiply-exposing a single frame in a camera.

RGB_GRAY_MAP

This atom names a property. The value of the property is an *XStandardColormap*.

The property describes the best *GrayScale* colormap available on the screen. As previously mentioned, only the colormap, red_max, red_mult and base_pixel members of the *XStandardColormap* structure are used for *GrayScale* colormaps.

15.3.2   Setting and Obtaining Standard Colormaps

Xlib provides functions that you can use to set and obtain an *XStandardColormap* structure.

To set an *XStandardColormap* structure, use *XSetRGBColormaps*.

```
void XSetRGBColormaps(display, w, std_colormap,
        count, property)
    Display *display;
    Window w;
    XStandardColormap *std_colormap;
    int count;
    Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

std_colormap   Specifies the *XStandardColormap* structure to be used.

count   Specifies the number of colormaps.

property   Specifies the property name.

The *XSetRGBColormaps* function replaces the RGB colormap definition in the specified property on the named window. If the property does not already exist, *XSetRGBColormaps* sets the RGB colormap definition in the specified property on the named window. The property is stored with a type of RGB_COLOR_MAP and a format of 32. Note that it is the caller's responsibility to honour the ICCCM restriction that only RGB_DEFAULT_MAP contain more than one definition.

The *XSetRGBColormaps* function usually is only used by window or session managers. To create a standard colormap, follow this procedure:

1. Open a new connection to the same server.

2. Grab the server.

3. See if the property is on the property list of the root window for the screen.

4. If the desired property is not present:

   — Create a colormap (unless using the default colormap of the screen).

   — Determine the color characteristics of the visual.

   — Call *XAllocColorPlanes* or *XAllocColorCells* to allocate cells in the colormap.

   — Call *XStoreColors* to store appropriate color values in the colormap.

   — Fill in the descriptive members in the *XStandardColormap* structure.

   — Attach the property to the root window.

   — Use *XSetCloseDownMode* to make the resource permanent.

5. Ungrab the server.

XSetRGBColormaps can generate *BadAlloc*, *BadAtom* and *BadWindow* errors.

To obtain the *XStandardColormap* structure associated with the specified property, use *XGetRGBColormaps*.

```
Status XGetRGBColormaps(display, w, std_colormap_return,
        count_return, property
    Display *display;
    Window w;
    XStandardColormap **std_colormap_return;
    int *count_return;
    Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

std_colormap_return

                   Returns the *XStandardColormap* structure.

count_return   Returns the number of colormaps.

property   Specifies the property name.

The *XGetRGBColormaps* function returns the RGB colormap definitions stored in the specified property on the named window. If the property exists, is of type RGB_COLOR_MAP, is of format 32, and is long enough to contain a colormap definition, *XGetRGBColormaps* allocates and fills in space for the returned colormaps and returns a non-zero status. If the visualid is not present, *XGetRGBColormaps* assumes the default visual for the screen on which the window is located; if the killid is not present, *None* is assumed, which indicates that the resources cannot be released. Otherwise, none of the fields are set, and *XGetRGBColormaps* returns a zero status. Note that it is the caller's responsibility to honour the ICCCM restriction that only RGB_DEFAULT_MAP contain more than one definition.

XGetRGBColormaps can generate *BadAtom* and *BadWindow* errors.

Chapter 16

Resource Manager Functions

A program often needs a variety of options in the X environment (for example, fonts, colors, icons and cursors). Specifying all of these options on the command line is awkward because users may want to customize many aspects of the program and need a convenient way to establish these customizations as the default setting. The resource manager is provided for this purpose. Resource specifications are usually stored in human-readable files and in server properties.

The resource manager is a database manager with a twist. In most database systems, you perform a query using an imprecise specification, and you get back a set of records. The resource manager, however, allows you to specify a large set of values with an imprecise specification, to query the database with a precise specification, and to get back only a single value. This should be used by applications that need to know what the user prefers for colors, fonts, and other resources. It is this use as a database for dealing with X resources that inspired the name ''Resource Manager'', although the resource manager can be and is used in other ways.

For example, a user of your application may want to specify that all windows should have a blue background but that all mail-reading windows should have a red background. With well-engineered and coordinated applications, a user can define this information using only two lines of specifications.

As an example of how the resource manager works, consider a mail-reading application called xmh. Assume that it is designed so that it uses a complex window hierarchy all the way down to individual command buttons, which may be actual small subwindows in some toolkits. These are often called objects or widgets. In such toolkit systems, each user interface object can be composed of other objects and can be assigned a name and a class. Fully qualified names or classes can have arbitrary numbers of component names, but a fully qualified name always has the same number of component names as a fully qualified class. This generally reflects the structure of the application as composed of these objects, starting with the application itself.

For example, the xmh mail program has a name ''xmh'' and is one of a class of ''Mail'' programs. By convention, the first character of class components is capitalized, and the first letter of name components is in lower case. Each name and class finally has an attribute (for example, ''foreground'' or ''font''). If each window is properly assigned a name and class, it is easy for the user to specify attributes of any portion of the application.

At the top level, the application might consist of a paned window (that is, a window divided into several sections) named ''toc''. One pane of the paned window is a button box window named ''buttons'' and is filled with command buttons. One of these command buttons is used to incorporate new mail and has the name ''incorporate''. This window has a fully qualified name, ''xmh.toc.buttons.incorporate'', and a fully qualified class, ''Xmh.Paned.Box.Command''. Its fully qualified name is the name of its parent, ''xmh.toc.buttons'', followed by its name, ''incorporate''. Its class is the class of its parent, ''Xmh.Paned.Box'', followed by its particular class, ''Command''. The fully qualified name of a resource is the attribute's name appended to the object's fully qualified name, and the fully qualified class is its class appended to the object's class.

The incorporate button might need the following resources: Title string, Font, Foreground color for its inactive state, Background color for its inactive state, Foreground color for its active state, and Background color for its active state. Each resource is considered to be an attribute of the button and, as such, has a name and a class. For example, the foreground color for the button in its active state might be named ''activeForeground'', and its class might be ''Foreground''.

When an application looks up a resource (for example, a color), it passes the complete name and complete class of the resource to a lookup routine. The resource manager compares this complete specification against the incomplete specifications of entries in the resource database, find the best match, and returns the corresponding value for that entry.

The definitions for the resource manager are contained in **<X11/Xresource.h>**.

16.1   Resource File Syntax

The syntax of a resource file is a sequence of resource lines terminated by newline characters or end of file. The syntax of an individual resource line is:

```
ResourceLine  =  Comment | IncludeFile | ResourceSpec | <empty line>
Comment       =  "!" {<any character except null or newline>}
IncludeFile   =  "#" WhiteSpace "include" WhiteSpace FileName WhiteSpace
FileName      =  <valid filename for operating system>
ResourceSpec  =  WhiteSpace ResourceName WhiteSpace ":" WhiteSpace Value
ResourceName  =  [Binding] {Component Binding} ComponentName
Binding       =  "." | "*"
WhiteSpace    =  {<space> | <horizontal tab>}
Component     =  "?" | ComponentName
ComponentName =  NameChar {NameChar}
NameChar      =  "a"-"z" | "A"-"Z" | "0"-"9" | "_" | "-"
Value         =  {<any character except null or unescaped newline>}
```

Elements separated by vertical bar (|) are alternatives. Curly braces ({...}) indicate zero or more repetitions of the enclosed elements. Square brackets ([...]) indicate that the enclosed element is optional. Quotes ("...") are used around literal characters.

IncludeFile lines are interpreted by replacing the line with the contents of the specified file. The word ''include'' must be in lower case. The filename is interpreted relative to the directory of the file in which the line occurs (for example, if the filename contains no directory or contains a relative directory specification).

If a ResourceName contains a contiguous sequence of two or more Binding characters, the sequence will be replaced with single ''.'' character if the sequence contains only ''.'' characters, otherwise the sequence will be replaced with a single ''*'' character.

A resource database never contains more than one entry for a given ResourceName. If a resource file contains multiple lines with the same ResourceName, the last line in the file is used.

Any whitespace character before or after the name or colon in a ResourceSpec are ignored. To allow a Value to begin with whitespace, the two-character sequence ''*space*'' (backslash followed by space) is recognized and replaced by a space character, and the two-character sequence ''*tab*'' (backslash followed by horizontal tab) is recognized and replaced by a horizontal tab character. To allow a Value to contain embedded newline characters, the two-character sequence ''\n'' is recognized and replaced by a newline character. To allow a Value to be broken across multiple lines in a text file, the two-character sequence ''*newline*'' (backslash followed by newline) is recognized and removed from the value. To allow a Value to contain arbitrary character codes, the four-character sequence ''*nnn*'', where each *n* is a digit character in the range of ''0''–''7'', is recognized and replaced with a single byte that contains the octal value specified by the sequence. Finally, the two-character sequence ''\\\\'' is recognized and replaced with a single backslash.

As an example of these sequences, the following resource line contains a value consisting of four characters: a backslash, a null, a ''z'', and a newline:

```
magic.values: \\\000\
z\n
```

16.2   Resource Manager Matching Rules

The algorithm for determining which resource database entry matches a given query is the heart of the resource manager. All queries must fully specify the name and class of the desired resource (use of "*" and "?" are not permitted). The library supports up to 100 components in a full name or class. Resources are stored in the database with only partially specified names and classes, using pattern matching constructs. An asterisk (*) is a loose binding and is used to represent any number of intervening components, including none. A period (.) is a tight binding and is used to separate immediately adjacent components. A question mark (?) is used to match any single component name or class. A database entry cannot end in a loose binding; the final component (which cannot be "?") must be specified. The lookup algorithm searches the database for the entry that most closely matches (is most specific for) the full name and class being queried. When more than one database entry matches the full name and class, precedence rules are used to select just one.

The full name and class are scanned from left to right (from highest level in the hierarchy to lowest), one component at a time. At each level, the corresponding component and/or binding of each matching entry is determined, and these matching components and bindings are compared according to precedence rules. Each of the rules is applied at each level, before moving to the next level, until a rule selects a single entry over all others. The rules (in order of precedence) are:

1. An entry that contains a matching component (whether name, class, or "?") takes precedence over entries that elide the level (that is, entries that match the level in a loose binding).

2. An entry with a matching name takes precedence over both entries with a matching class and entries that match using "?". An entry with a matching class takes precedence over entries that match using "?".

3. An entry preceded by a tight binding takes precedence over entries preceded by a loose binding.

To illustrate these rules, consider following the resource database entries:

```
xmh*Paned*activeForeground:        red      (entry A)
*incorporate.Foreground:           blue     (entry B)
xmh.toc*Command*activeForeground:  green    (entry C)
xmh.toc*?.Foreground:              white    (entry D)
xmh.toc*Command.activeForeground:  black    (entry E)
```

Consider a query for the resource:

```
xmh.toc.messagefunctions.incorporate.activeForeground (name)
Xmh.Paned.Box.Command.Foreground                      (class)
```

At the first level (xmh, Xmh) rule 1 eliminates entry B. At the second level (toc, Paned) rule 2 eliminates entry A. At the third level (messagefunctions, Box) no entries are eliminated. At the fourth level (incorporate, Command) rule 2 eliminates entry D. At the fifth level (activeForeground, Foreground) rule 3 eliminates entry C.

16.3 Quarks

Most uses of the resource manager involve defining names, classes, and representation types as string constants. However, always referring to strings in the resource manager can be slow, because it is so heavily used in some toolkits. To solve this problem, a shorthand for a string is used in place of the string in many of the resource manager functions. Simple comparisons can be performed rather than string comparisons. The shorthand name for a string is called a quark and is the type *XrmQuark*. On some occasions, you may want to allocate a quark that has no string equivalent.

A quark is to a string what an atom is to a string in the server, but its use is entirely local to your application.

To allocate a new quark, use *XrmUniqueQuark*.

```
XrmQuark XrmUniqueQuark()
```

The *XrmUniqueQuark* function allocates a quark that is guaranted not to represent any string that is known to the resource manager.

Each name, class and representation type is typedef'd as an *XrmQuark*.

```
typedef int XrmQuark, *XrmQuarkList;
typedef XrmQuark XrmName;
typedef XrmQuark XrmClass;
typedef XrmQuark XrmRepresentation;
#define NULLQUARK ((XrmQuark) 0)
```

Lists are represented as null-terminated arrays of quarks. The size of the array must be large enough for the number of components used.

```
typedef XrmQuarkList XrmNameList;
typedef XrmQuarkList XrmClassList;
```

To convert a string to a quark, use *XrmStringToQuark* or *XrmPermStringToQuark*.

```
#define XrmStringToName(string) XrmStringToQuark(string)
#define XrmStringToClass(string) XrmStringToQuark(string)
#define XrmStringToRepresentation(string) XrmStringToQuark(string)
```

```
XrmQuark XrmStringToQuark(string)
      char *string;
```

```
XrmQuark XrmPermStringToQuark(string)
      char *string;
```

string   Specifies the string for which a quark is to be allocated.

These functions can be used to convert from string to quark representation. If the string is not in the Host Portable Character Encoding the conversion is implementation-dependent. The string argument to *XrmStringToQuark* need not be permanently allocated storage. *XrmPermStringToQuark* is just like *XrmStringToQuark*, except that Xlib is permitted to assume the string argument is permanently allocated, and hence that it can be used as the value to be returned by *XrmQuarkToString*.

To convert a quark to a string, use *XrmQuarkToString*.

```
#define XrmNameToString(name) XrmQuarkToString(name)
#define XrmClassToString(class) XrmQuarkToString(class)
#define XrmRepresentationToString(type) XrmQuarkToString(type)

char *XrmQuarkToString(quark)
     XrmQuark quark;
```

quark   Specifies the quark for which the equivalent string is desired.

This function can be used to convert from quark representation to string. The string pointed to by the return value must not be modified or freed. The returned string is byte-for-byte equal to the original string passed to one of the string-to-quark routines. If no string exists for that quark, *XrmQuarkToString* returns NULL. For any given quark, if *XrmQuarkToString* returns a non-NULL value, all future calls will return the same value (identical address).

To convert a string with one or more components to a quark list, use *XrmStringToQuarkList*.

```
#define XrmStringToNameList(str, name)  XrmStringToQuarkList((str), (name))
#define XrmStringToClassList(str,class) XrmStringToQuarkList((str), (class))

void XrmStringToQuarkList(string, quarks_return)
     char *string;
     XrmQuarkList quarks_return;
```

string   Specifies the string for which a quark list is to be allocated.

quarks_return   Returns the list of quarks.

The *XrmStringToQuarkList* function converts the null-terminated string (generally a fully qualified name) to a list of quarks. Note that the string must be in the valid ResourceName format (see Section 16.1). If the string is not in the Host Portable Character Encoding the conversion is implementation-dependent.

A binding list is a list of type *XrmBindingList* and indicates if components of name or class lists are bound tightly or loosely (that is, if wildcarding of intermediate components is specified).

```
typedef enum {XrmBindTightly, XrmBindLoosely} XrmBinding, *XrmBindingList;
```

XrmBindTightly indicates that a period separates the components, and *XrmBindLoosely* indicates that an asterisk separates the components.

To convert a string with one or more components to a binding list and a quark list, use *XrmStringToBindingQuarkList*.

```
XrmStringToBindingQuarkList(string, bindings_return,
        quarks_return)
     char *string;
     XrmBindingList bindings_return;
     XrmQuarkList quarks_return;
```

string   Specifies the string for which a quark list is to be allocated.

bindings_return Returns the binding list. The caller must allocate sufficient space for the binding list before calling *XrmStringToBindingQuarkList*.

quarks_return   Returns the list of quarks. The caller must allocate sufficient space for the quarks list before calling *XrmStringToBindingQuarkList*.

Component names in the list are separated by a period or an asterisk character. The string must be in the format of a valid ResourceName (see Section 16.1). If the string does not start with a

period or an asterisk, a tight binding is assumed. For example, ''*a.b*c'' becomes:

```
quarks:    a        b        c
bindings:  loose    tight    loose
```

16.4   Creating and Storing Databases

A resource database is an opaque type, *XrmDatabase*. Each database value is stored in an *XrmValue* structure. This structure consists of a size, an address, and a representation type. The size is specified in bytes. The representation type is a way for you to store data tagged by some application-defined type (for example, ''font'' or ''color''). It has nothing to do with the C data type or with its class. The *XrmValue* structure is defined as:

```
typedef struct {
      unsigned int size;
      XPointer addr;
} XrmValue, *XrmValuePtr;
```

To initialize the resource manager, use *XrmInitialize*.

```
void XrmInitialize();
```

To retrieve a database from disk, use *XrmGetFileDatabase*.

```
XrmDatabase XrmGetFileDatabase(filename)
      char *filename;
```

filename   Specifies the resource database file name.

The *XrmGetFileDatabase* function opens the specified file, creates a new resource database, and loads it with the specifications read in from the specified file. The specified file must contain a sequence of entries in valid ResourceLine format (see Section 16.1). The file is parsed in the current locale, and the database is created in the current locale. If it cannot open the specified file, *XrmGetFileDatabase* returns NULL.

To store a copy of a database to disk, use *XrmPutFileDatabase*.

```
void XrmPutFileDatabase(database, stored_db)
      XrmDatabase database;
      char *stored_db;
```

database   Specifies the database that is to be used.

stored_db   Specifies the file name for the stored database.

The *XrmPutFileDatabase* function stores a copy of the specified database in the specified file. Text is written to the file as a sequence of entries in valid ResourceLine format (see Section 16.1). The file is written in the locale of the database. Entries containing resource names that are not in the Host Portable Character Encoding, or containing values that are not in the encoding of the database locale, are written in an implementation-dependent manner. The order in which entries are written is implementation-dependent. Entries with representation types other than ''String'' are ignored.

To obtain a pointer to the screen-independent resources of a display, use *XResourceManagerString*.

```
char *XResourceManagerString(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XResourceManagerString* returns the RESOURCE_MANAGER property from the server's root window of screen zero, which was returned when the connection was opened using *XOpenDisplay*. The property is converted from type STRING to the current locale. The conversion is identical to that produced by *XmbTextPropertyToTextList* for a singleton STRING property. The returned string is owned by Xlib, and should not be freed by the client. Note that

the property value must be in a format that is acceptable to *XrmGetStringDatabase*. If no
property exists, NULL is returned.

To obtain a pointer to the screen-specific resources of a screen, use *XScreenResourceString*.

```
char *XScreenResourceString(screen)
      Screen *screen;
```

screen   Specifies the screen.

The *XStringResourceString* returns the SCREEN_RESOURCES property from the root window of
the specified screen. The property is converted from type STRING to the current locale. The
conversion is identical to that produced by *XmbTextPropertyToTextList* for a singleton STRING
property. Note that the property value must be in a format that is acceptable to
XrmGetStringDatabase. If no property exists, NULL is returned. The caller is responsible for
freeing the returned string, using *XFree*.

To create a database from a string, use *XrmGetStringDatabase*.

```
XrmDatabase XrmGetStringDatabase(data)
      char *data;
```

data   Specifies the database contents using a string.

The *XrmGetStringDatabase* function creates a new database and stores the resources specified in
the specified null-terminated string. *XrmGetStringDatabase* is similar to *XrmGetFileDatabase*
except that it reads the information out of a string instead of out of a file. The string must
contain a sequence of entries in valid ResourceLine format (see Section 16.1). The string is
parsed in the current locale, and the database is created in the current locale.

To obtain locale name of a database, use *XrmLocaleOfDatabase*.

```
char *XrmLocaleOfDatabase(database)
      XrmDatabase database;
```

database   Specifies the resource database.

The *XrmLocaleOfDatabase* function returns the name of the locale bound to the specified
database, as a null-terminated string. The returned locale name string is owned by Xlib and
should not be modified or freed by the client. Xlib is not permitted to free the string until the
database is destroyed. Until the string is freed, it will not be modified by Xlib.

To destroy a resource database and free its allocated memory, use *XrmDestroyDatabase*.

```
void XrmDestroyDatabase(database)
      XrmDatabase database;
```

database   Specifies the resource database.

If database is NULL, *XrmDestroyDatabase* returns immediately.

To associate a resource database with a display, use *XrmSetDatabase*.

```
void XrmSetDatabase(display, database)
      Display *display;
      XrmDatabase database;
```

display   Specifies the connection to the X server.

database   Specifies the resource database.

The *XrmSetDatabase* function associates the specified resource database (or NULL) with the
specified display. The database previously associated with the display (if any) is not destroyed.

A client or toolkit may find this function convenient for retaining a database once it is constructed.

To get the resource database associated with a display, use *XrmGetDatabase*.

```
XrmDatabase XrmGetDatabase(display)
      Display *display;
```

display   Specifies the connection to the X server.

The *XrmGetDatabase* function returns the database associated with the specified display. It returns NULL if a database has not yet been set.

16.5   Merging Resource Databases

To merge the contents of a resource file into a database, us *XrmCombineFileDatabase*.

```
void XrmCombineFileDatabase(filename, target_db, override)
      char *filename;
      XrmDatabase *target_db;
      Bool override;
```

filename   Specifies the resource database file name.

target_db   Specifies the resource database into which the source database is to be merged.

The *XrmCombineFileDatabase* function merges the contents of a resource file into a database. If the same specifier is used for an entry in both the file and the database, the entry in the file will replace the entry in the database if override is *True*; otherwise, the entry in file is discarded. The file is parsed in the current locale. If the file cannot be read a zero status is returned; otherwise a nonzero status is returned. If target_db contains NULL, *XrmCombineFileDatabase* creates and returns a new database to it. Otherwise, the database pointed to by target_db is not destroyed by the merge. The database entries are merged without changing values or types, regardless of the locale of the database. The locale of the target database is not modified.

To merge the contents of one database into another database, use *XrmCombineDatabase*.

```
void XrmCombineDatabase(source_db, target_db, override)
      XrmDatabase source_db, *target_db;
      Bool override;
```

source_db   Specifies the resource database that is to be merged into the target database.

target_db   Specifies the resource database into which the source database is to be merged.

override   Specifies whether source entries override target ones.

The *XrmCombineDatabase* function merges the contents of one database into another. If the same specifier is used for an entry in both databases, the entry in the source_db will replace the entry in the target_db if override is *True*; otherwise, the entry in source_db is discarded. If target_db contains NULL, *XrmCombineDatabase* simply stores source_db in it. Otherwise, source_db is destroyed by the merge, but the database pointed to by target_db is not destroyed. The database entries are merged without changing values or types, regardless of the locales of the databases. The locale of the target database is not modified.

To merge the contents of one database into another database with override semantics, use *XrmMergeDatabases*.

```
void XrmMergeDatabases(source_db, target_db)
      XrmDatabase source_db, *target_db;
```

source_db   Specifies the resource database that is to be merged into the target database.

target_db   Specifies the resource database into which the source database is to be merged.

The *XrmMergeDatabases* function merges the contents of one database into another. If the same specifier is used for an entry in both databases, the entry in the source_db will replace the entry in the target_db (that is, it overrides target_db). If target_db contains NULL, *XrmMergeDatabases* simply stores source_db in it. Otherwise, source_db is destroyed by the merge, but the database pointed to by target_db is not destroyed. The database entries are merged without changing

values or types, regardless of the locales of the databases. The locale of the target database is not
modified.

16.6   Looking Up Resources

To retrieve a resource from a resource database, use *XrmGetResource*, *XrmQGetResource* or *XrmQGetSearchResource*.

```
Bool XrmGetResource(database, str_name, str_class,
        str_type_return, value_return)
    XrmDatabase database;
    char *str_name;
    char *str_class;
    char **str_type_return;
    XrmValue *value_return;
```

database	Specifies the database that is to be used.
str_name	Specifies the fully qualified name of the value being retrieved (as a string).
str_class	Specifies the fully qualified class of the value being retrieved (as a string).
str_type_return	Returns the representation type of the destination (as a string).
value_return	Returns the value in the database.

```
Bool XrmQGetResource(database, quark_name, quark_class,
        quark_type_return, value_return)
    XrmDatabase database;
    XrmNameList quark_name;
    XrmClassList quark_class;
    XrmRepresentation *quark_type_return;
    XrmValue *value_return;
```

database	Specifies the database that is to be used.
quark_name	Specifies the fully qualified name of the value being retrieved (as a quark).
quark_class	Specifies the fully qualified class of the value being retrieved (as a quark).
quark_type_return	Returns the representation type of the destination (as a quark).
value_return	Returns the value in the database.

The *XrmGetResource* and *XrmQGetResource* functions retrieve a resource from the specified database. Both take a fully qualified name/class pair, a destination resource representation, and the address of a value (size/address pair). The value and returned type point into database memory; therefore, you must not modify the data.

The database only frees or overwrites entries on *XrmPutResource*, *XrmQPutResource* or *XrmMergeDatabases*. A client that is not storing new values into the database or is not merging the database should be safe using the address passed back at any time until it exits. If a resource was found, both *XrmGetResource* and *XrmQGetResource* return *True*; otherwise, they return *False*.

Most applications and toolkits do not make random probes into a resource database to fetch resources. The X toolkit access pattern for a resource database is quite stylized. A series of from 1 to 20 probes are made with only the last name/class differing in each probe. The *XrmGetResource* function is at worst a 2^n algorithm, where *n* is the length of the name/class list. This can be improved upon by the application programmer by prefetching a list of database levels that might match the first part of a name/class list.

To obtain a list of database levels, use *XrmQGetSearchList*.

```
        typedef XrmHashTable *XrmSearchList;

    Bool XrmQGetSearchList(database, names, classes, list_return, list_length)
        XrmDatabase database;
        XrmNameList names;
        XrmClassList classes;
        XrmSearchList list_return;
        int list_length;
```

database   Specifies the database that is to be used.

names   Specifies a list of resource names.

classes   Specifies a list of resource classes.

list_return   Returns a search list for further use. The caller must allocate sufficient space
                    for the list before calling *XrmQGetSearchList*.

list_length   Specifies the number of entries (not the byte size) allocated for list_return.

The *XrmQGetSearchList* function takes a list of names and classes and returns a list of database
levels where a match might occur. The returned list is in best-to-worst order and uses the same
algorithm as *XrmGetResource* for determining precedence. If list_return was large enough for the
search list, *XrmQGetSearchList* returns *True*; otherwise, it returns *False*.

The size of the search list that the caller must allocate is dependent upon the number of levels
and wildcards in the resource specifiers that are stored in the database. The worst case length is
3^n, where *n* is the number of name or class components in names or classes.

When using *XrmQGetSearchList* followed by multiple probes for resources with a common name
and class prefix, only the common prefix should be specified in the name and class list to
XrmQGetSearchList.

To search resource database levels for a given resource, use *XrmQGetSearchResource*.

```
    Bool XrmQGetSearchResource(list, name, class,
            type_return, value_return)
        XrmSearchList list;
        XrmName name;
        XrmClass class;
        XrmRepresentation *type_return;
        XrmValue *value_return;
```

list   Specifies the search list returned by *XrmQGetSearchList*.

name   Specifies the resource name.

class   Specifies the resource class.

type_return   Returns data representation type.

value_return   Returns the value in the database.

The *XrmQGetSearchResource* function searches the specified database levels for the resource that
is fully identified by the specified name and class. The search stops with the first match.
XrmQGetSearchResource returns *True* if the resource was found; otherwise, it returns *False*.

A call to *XrmQGetSearchList* with a name and class list containing all but the last component of a
resource name followed by a call to *XrmQGetSearchResource* with the last component name and
class returns the same database entry as *XrmGetResource* and *XrmQGetResource* with the fully
qualified name and class.

16.7 Storing Into a Resource Database

To store resources into the database, use *XrmPutResource* or *XrmQPutResource*. Both functions take a partial resource specification, a representation type, and a value. This value is copied into the specified database.

```
void XrmPutResource(database, specifier, type, value)
    XrmDatabase *database;
    char *specifier;
    char *type;
    XrmValue *value;
```

database   Specifies the resource database.

specifier   Specifies a complete or partial specification of the resource.

type   Specifies the type of the resource.

value   Specifies the value of the resource, which is specified as a string.

If database contains NULL, *XrmPutResource* creates a new database and returns a pointer to it. *XrmPutResource* is a convenience function that calls *XrmStringToBindingQuarkList* followed by:

```
XrmQPutResource(database, bindings, quarks, XrmStringToQuark(type), value)
```

If the specifier and type are not in the Host Portable Character Encoding the result is implementation-dependent. The value is stored in the database without modification.

```
void XrmQPutResource(database, bindings, quarks, type,
        value)
     XrmDatabase *database;
     XrmBindingList bindings;
     XrmQuarkList quarks;
     XrmRepresentation type;
     XrmValue *value;
```

database   Specifies the resource database.

bindings   Specifies a list of bindings.

quarks   Specifies the complete or partial name or the class list of the resource.

type   Specifies the type of the resource.

value   Specifies the value of the resource, which is specified as a string.

If database contains NULL, *XrmQPutResource* creates a new database and returns a pointer to it. If a resource entry with the identical bindings and quarks already exists in the database, the previous value is replaced by the new specified value. The value is stored in the database without modification.

To add a resource that is specified as a string, use *XrmPutStringResource*.

```
void XrmPutStringResource(database, specifier, value)
    XrmDatabase *database;
    char *specifier;
    char *value;
```

database   Specifies the resource database.

specifier   Specifies a complete or partial specification of the resource.

value   Specifies the value of the resource, which is specified as a string.

If database contains NULL, *XrmPutStringResource* creates a new database and returns a pointer to it. *XrmPutStringResource* adds a resource with the specified value to the specified database. *XrmPutStringResource* is a convenience function that first calls *XrmStringToBindingQuarkList* on the specifier and then calls *XrmQPutResource*, using a ''String'' representation type. If the specifier is not in the Host Portable Character Encoding the result is implementation-dependent. The value is stored in the database without modification.

To add a string resource using quarks as a specification, use *XrmQPutStringResource*.

```
void XrmQPutStringResource(database, bindings, quarks, value)
    XrmDatabase *database;
    XrmBindingList bindings;
    XrmQuarkList quarks;
    char *value;
```

database   Specifies the resource database.

bindings   Specifies a list of bindings.

quarks   Specifies the complete or partial name or the class list of the resource.

value   Specifies the value of the resource, which is specified as a string.

If database contains NULL, *XrmQPutStringResource* creates a new database and returns a pointer to it. *XrmQPutStringResource* is a convenience routine that constructs an *XrmValue* for the value string (by calling *strlen* to compute the size) and then calls *XrmQPutResource*, using a ''String'' representation type. The value is stored in the database without modification.

To add a single resource entry that is specified as a string that contains both a name and a value, use *XrmPutLineResource*.

```
void XrmPutLineResource(database, line)
     XrmDatabase *database;
     char *line;
```

database   Specifies the resource database.

line   Specifies the resource name and value pair as a single string.

If database contains NULL, *XrmPutLineResource* creates a new database and returns a pointer to it. *XrmPutLineResource* adds a single resource entry to the specified database. The line must be in valid ResourceLine format (see Section 16.1). The string is parsed in the locale of the database. If the *ResourceName* is not in the Host Portable Character Encoding the result is implementation-dependent. Note that comment lines are not stored.

16.8   Enumerating Database Entries

To enumerate the entries of a database, use *XrmEnumerateDatabase*.

```
#define    XrmEnumAllLevels    0
#define    XrmEnumOneLevel     1

Bool XrmEnumerateDatabase(database, name_prefix, class_prefix,
         mode, proc, arg)
      XrmDatabase database;
      XrmNameList name_prefix;
      XrmClassList class_prefix;
      int mode;
      Bool (*proc)();
      XPointer arg;
```

database	Specifies the resource database.
name_prefix	Specifies the resource name prefix.
class_prefix	Specifies the resource class prefix.
mode	Specifies the number of levels to enumerate.
proc	Specifies the procedure that is to be called for each matching entry.
arg	Specifies the user-supplied argument that will be passed to the procedure.

The *XrmEnumerateDatabase* function calls the specified procedure for each resource in the database that would match some completion of the given name/class resource prefix. The order in which resources are found is implementation-dependent. If mode is *XrmEnumOneLevel*, then a resource must match the given name/class prefix with just a single name and class appended. If mode is *XrmEnumAllLevels*, the resource must match the given name/class prefix with one or more names and classes appended. If the procedure returns *True*, the enumeration terminates and the function returns *True*. If the procedure always returns *False*, all matching resources are enumerated and the function returns *False*.

The procedure is called with the following arguments:

```
(*proc)(database, bindings, quarks, type,
         value, arg)
      XrmDatabase *database;
      XrmBindingList bindings;
      XrmQuarkList quarks;
      XrmRepresentation *type;
      XrmValue *value;
      XPointer closure;
```

The bindings and quarks lists are terminated by *NULLQUARK*. Note that pointers to the database and type are passed, but these values should not be modified.

16.9   Parsing Command Line Options

The *XrmParseCommand* function can be used to parse the command line arguments to a program and modify a resource database with selected entries from the command line.

```
typedef enum {
        XrmoptionNoArg,     /* Value is specified in
                                 XrmOptionDescRec.value */
        XrmoptionIsArg,     /* Value is the option string itself */
        XrmoptionStickyArg, /* Value is characters immediately
                                 following option */
        XrmoptionSepArg,    /* Value is next argument in argv */
        XrmoptionResArg,    /* Resource and value in next argument
                                 in argv */
        XrmoptionSkipArg,   /* Ignore this option and the next
                                 argument in argv */
        XrmoptionSkipLine,  /* Ignore this option and the rest of argv */
        XrmoptionSkipNArgs  /* Ignore this option and the next
                                 XrmOptionDescRec.value arguments in argv */
} XrmOptionKind;
```

Note that *XrmoptionSkipArg* is equivalent to *XrmoptionSkipNArgs* with the *XrmOptionDescRec.value* field containing the value one. Note also that the value zero for *XrmoptionSkipNArgs* indicates that only the option itself is to be skipped.

```
typedef struct {
        char *option;          /* Option specification string in argv */
        char *specifier;       /* Binding and resource name (sans
                                   application name) */
        XrmOptionKind argKind; /* Which style of option it is */
        XPointer value;        /* Value to provide if XrmoptionNoArg or
                                   XrmoptionSkipNArgs */
} XrmOptionDescRec, *XrmOptionDescList;
```

To load a resource database from a C command line, use *XrmParseCommand*.

```
void XrmParseCommand(database, table, table_count, name,
        argc_in_out, argv_in_out)
        XrmDatabase *database;
        XrmOptionDescList table;
        int table_count;
        char *name;
        int *argc_in_out;
        char **argv_in_out;
```

database   Specifies the resource database.

table   Specifies the table of command line arguments to be parsed.

table_count   Specifies the number of entries in the table.

name   Specifies the application name.

argc_in_out   Specifies the number of arguments and returns the number of remaining arguments.

argv_in_out   Specifies the command line arguments and returns the remaining arguments.

The *XrmParseCommand* function parses an (argc, argv) pair according to the specified option table, loads recognized options into the specified database with type ''String,'' and modifies the (argc, argv) pair to remove all recognized options. If database contains NULL, *XrmParseCommand* creates a new database and returns a pointer to it. Otherwise, entries are added to the database specified. If a database is created, it is created in the current locale.

The specified table is used to parse the command line. Recognized options in the table are removed from argv, and entries are added to the specified resource database. The table entries contain information on the option string, the option name, the style of option, and a value to provide if the option kind is *XrmoptionNoArg*. The option names are compared byte-for-byte to arguments in argv, independent of any locale. The resource values given in the table are stored in the resource database without modification. All resource database entries are created using a ''String'' representation type. The argc argument specifies the number of arguments in argv and is set on return to the remaining number of arguments that were not parsed. The name argument should be the name of your application for use in building the database entry. The name argument is prefixed to the resourceName in the option table before storing a database entry. No separating (binding) character is inserted, so the table must contain either a period (.) or an asterisk (*) as the first character in each resourceName entry. To specify a more completely qualified resource name, the resourceName entry can contain multiple components. If the name argument and the resourceNames are not in the Host Portable Character Encoding the result is implementation-dependent.

The following provides a sample option table:

```
static XrmOptionDescRec opTable[] = {
{"-background",  "*background",                XrmoptionSepArg, (XPointer) NULL},
{"-bd",          "*borderColor",               XrmoptionSepArg, (XPointer) NULL},
{"-bg",          "*background",                XrmoptionSepArg, (XPointer) NULL},
{"-borderwidth", "*TopLevelShell.borderWidth", XrmoptionSepArg, (XPointer) NULL},
{"-bordercolor", "*borderColor",               XrmoptionSepArg, (XPointer) NULL},
{"-bw",          "*TopLevelShell.borderWidth", XrmoptionSepArg, (XPointer) NULL},
{"-display",     ".display",                   XrmoptionSepArg, (XPointer) NULL},
{"-fg",          "*foreground",                XrmoptionSepArg, (XPointer) NULL},
{"-fn",          "*font",                      XrmoptionSepArg, (XPointer) NULL},
{"-font",        "*font",                      XrmoptionSepArg, (XPointer) NULL},
{"-foreground",  "*foreground",                XrmoptionSepArg, (XPointer) NULL},
{"-geometry",    ".TopLevelShell.geometry",    XrmoptionSepArg, (XPointer) NULL},
{"-iconic",      ".TopLevelShell.iconic",      XrmoptionNoArg,  (XPointer) "on"},
{"-name",        ".name",                      XrmoptionSepArg, (XPointer) NULL},
{"-reverse",     "*reverseVideo",              XrmoptionNoArg,  (XPointer) "on"},
{"-rv",          "*reverseVideo",              XrmoptionNoArg,  (XPointer) "on"},
{"-synchronous", "*synchronous",               XrmoptionNoArg,  (XPointer) "on"},
{"-title",       ".TopLevelShell.title",       XrmoptionSepArg, (XPointer) NULL},
{"-xrm",         NULL,                         XrmoptionResArg, (XPointer) NULL},
};
```

In this table, if the –background (or –bg) option is used to set background colors, the stored resource specifier matches all resources of attribute background. If the –borderwidth option is used, the stored resource specifier applies only to border width attributes of class TopLevelShell (that is, outer-most windows, including pop-up windows). If the –title option is used to set a window name, only the topmost application windows receive the resource.

When parsing the command line, any unique unambiguous abbreviation for an option name in the table is considered a match for the option. Note that upper case and lower case matter.

Application Utility Functions

Once you have initialized the X system, you can use the Xlib utility functions to:

- obtain and classify KeySyms
- allocate permanent storage
- parse window geometry strings
- manipulate regions
- use cut buffers
- determine the appropriate visual
- manipulate images
- manipulate bitmaps
- Use the context manager.

As a group, the functions discussed in this chapter provide the functionality that is frequently needed and that spans toolkits. Many of these functions do not generate actual protocol requests to the server.

17.1   Keyboard Utility Functions

This section discusses mapping between KeyCodes and KeySyms, names for KeySyms, and KeySym classification macros. The functions in this section operate on an cached copy of the server keyboard mapping. The first four KeySyms for each key code are modified according to the rules given in Section 13.7. If you want the untransformed KeySyms defined for a key, you should only use the functions described in Section 13.7.

To obtain a KeySym for the key code of an event, use *XLookupKeysym*.

```
KeySym XLookupKeysym(key_event, index)
      XKeyEvent *key_event;
      int index;
```

key_event   Specifies the *KeyPress* or *KeyRelease* event.

index   Specifies the index into the KeySyms list for the event's KeyCode.

The *XLookupKeysym* function uses a given keyboard event and the index you specified to return the KeySym from the list that corresponds to the KeyCode member in the *XKeyPressedEvent* or *XKeyReleasedEvent* structure. If no KeySym is defined for the KeyCode of the event, *XLookupKeysym* returns *NoSymbol*.

To obtain a KeySym for a specific key code, use *XKeycodeToKeysym*.

```
KeySym XKeycodeToKeysym(display, keycode, index)
      Display *display;
      KeyCode keycode;
      int index;
```

display   Specifies the connection to the X server.

keycode   Specifies the KeyCode.

index   Specifies the element of KeyCode vector.

The *XKeycodeToKeysym* function uses internal Xlib tables and returns the KeySym defined for the specified KeyCode and the element of the KeyCode vector. If no symbol is defined, *XKeycodeToKeysym* returns *NoSymbol*.

To obtain a key code for a key having a specific KeySym, use *XKeysymToKeycode*.

```
KeyCode XKeysymToKeycode(display, keysym)
      Display *display;
      KeySym keysym;
```

display   Specifies the connection to the X server.

keysym   Specifies the KeySym that is to be searched for.

If the specified KeySym is not defined for any KeyCode, *XKeysymToKeycode* returns zero.

The mapping between key codes and KeySyms is cached internal to Xlib. When this information is changed at the server, an Xlib function must be called to refresh the cache. To refresh the stored modifier and keymap information, use *XRefreshKeyboardMapping*.

```
XRefreshKeyboardMapping(event_map)
      XMappingEvent *event_map;
```

event_map   Specifies the mapping event that is to be used.

The *XRefreshKeyboardMapping* function refreshes the stored modifier and keymap information. You usually call this function when a *MappingNotify* event with a request member of

MappingKeyboard or *MappingModifier* occurs. The result is to update Xlib's knowledge of the keyboard.

KeySyms have string names as well as numeric codes. To convert the name of the KeySym to the KeySym code, use *XStringToKeysym*.

```
KeySym XStringToKeysym(string)
      char *string;
```

string   Specifies the name of the KeySym that is to be converted.

Standard KeySym names are obtained from **<X11/keysymdef.h>** by removing the XK_ prefix from each name. KeySyms that are not part of the Xlib standard also may be obtained with this function. Note that the set of KeySysms that are available in this manner and the mechanisms by which Xlib obtains them is implementation-dependent.

If the keysym name is not in the Host Portable Character Encoding the result is implementation-dependent. If the specified string does not match a valid KeySym, *XStringToKeysym* returns *NoSymbol*.

To convert a KeySym code to the name of the KeySym, use *XKeysymToString*.

```
char *XKeysymToString(keysym)
      KeySym keysym;
```

keysym   Specifies the KeySym that is to be converted.

The returned string is in a static area and must not be modified. The returned string is in the Host Portable Character Encoding. If the specified KeySym is not defined, *XKeysymToString* returns a NULL.

17.1.1   Keysym Classification Macros

You may want to test if a KeySym is, for example, on the keypad or on one of the function keys. You can use the KeySym macros to perform the following tests.

```
IsCursorKey(keysym)
```

keysym   Specifies the KeySym that is to be tested.

Returns *True* if the specified KeySym is a cursor key.

```
IsFunctionKey(keysym)
```

keysym   Specifies the KeySym that is to be tested.

Returns *True* if the specified KeySym is a function key.

```
IsKeypadKey(keysym)
```

keysym   Specifies the KeySym that is to be tested.

Returns *True* if the specified KeySym is a keypad key.

```
IsMiscFunctionKey(keysym)
```

keysym   Specifies the KeySym that is to be tested.

Returns *True* if the specified KeySym is a miscellaneous function key.

```
IsModifierKey(keysym)
```

keysym   Specifies the KeySym that is to be tested.

Returns *True* if the specified KeySym is a modifier key.

```
IsPFKey(keysym)
```

keysym   Specifies the KeySym that is to be tested.

Returns *True* if the specified KeySym is a PF key.

17.2   **Latin-1 Keyboard Event Functions**

Chapter 13 describes internationalized text input facilities, but sometimes it is expedient to write an application that only deals with Latin-1 characters and ASCII controls, so Xlib provides a simple function for that purpose. *XLookupString* handles the standard modifier semantics described in Section 13.7. This function does not use any of the input method facilities described in Chapter 13, and does not depend on the current locale.

To map a key event to an ISO Latin-1 string, use *XLookupString*.

```
int XLookupString(event_struct, buffer_return, bytes_buffer,
        keysym_return, status_in_out)
    XKeyEvent *event_struct;
    char *buffer_return;
    int bytes_buffer;
    KeySym *keysym_return;
    XComposeStatus *status_in_out;
```

event_struct   Specifies the key event structure to be used. You can pass *XKeyPressedEvent* or *XKeyReleasedEvent*.

buffer_return   Returns the translated characters.

bytes_buffer   Specifies the length of the buffer. No more than bytes_buffer of translation are returned.

keysym_return   Returns the KeySym computed from the event if this argument is not NULL.

status_in_out   Specifies or returns the *XComposeStatus* structure or NULL.

The *XLookupString* function translates a key event to a KeySym and a string. The KeySym is obtained by using the standard interpretation of the Shift, Lock, and group modifiers as defined in the **X Window System Protocol** specification. If the KeySym has been rebound (see *XRebindKeysym*), the bound string will be stored in the buffer. Otherwise, the KeySym is mapped, if possible, to an ISO Latin-1 character or (if the Control modifier is on) to an ASCII control character, and that character is stored in the buffer. *XLookupString* returns the number of characters that are stored in the buffer.

If present (non-NULL), the *XComposeStatus* structure records the state, which is private to Xlib, that needs preservation across calls to *XLookupString* to implement compose processing. The creation of *XComposeStatus* structures is implementation-dependent; a portable program must pass NULL for this argument.

XLookupString depends on the cached keyboard information mentioned in the previous section, so it is necessary to use *XRefreshKeyboardMapping* to keep this information up to date.

To rebind the meaning of a KeySym for *XLookupString*, use *XRebindKeysym*.

```
XRebindKeysym(display, keysym, list, mod_count,
        string, num_bytes)
    Display *display;
    KeySym keysym;
    KeySym list[];
    int mod_count;
    unsigned char *string;
    int num_bytes;
```

display   Specifies the connection to the X server.

keysym	Specifies the KeySym that is to be rebound.
list	Specifies the KeySyms to be used as modifiers.
mod_count	Specifies the number of modifiers in the modifier list.
string	Specifies the string that is copied and will be returned by *XLookupString*.
num_bytes	Specifies the number of bytes in the string argument.

The *XRebindKeysym* function can be used to rebind the meaning of a KeySym for the client. It does not redefine any key in the X server but merely provides an easy way for long strings to be attached to keys. *XLookupString* returns this string when the appropriate set of modifier keys are pressed and when the KeySym would have been used for the translation. No text conversions are performed; the client is responsible for supplying appropriately encoded strings. Note that you can rebind a KeySym that may not exist.

17.3   Allocating Permanent Storage

To allocate some memory you will never give back, use *Xpermalloc*.

```
char *Xpermalloc(size)
      unsigned int size;
```

The *Xpermalloc* function allocates storage that can never be freed for the life of the program. The memory is allocated with alignment for the C type double. This function may provide some performance and space savings over the standard operating system memory allocator.

17.4   Parsing the Window Geometry

To parse standard window geometry strings, use *XParseGeometry*.

```
int XParseGeometry(parsestring, x_return, y_return,
        width_return, height_return)
    char *parsestring;
    int *x_return, *y_return;
    unsigned int *width_return, *height_return;
```

parsestring   Specifies the string you want to parse.

x_return
y_return   Return the x and y offsets.

width_return
height_return   Return the width and height determined.

By convention, X applications use a standard string to indicate window size and placement. *XParseGeometry* makes it easier to conform to this standard because it allows you to parse the standard window geometry. Specifically, this function lets you parse strings of the form:

```
[=][<width>{xX}<height>][{+-}<xoffset>{+-}<yoffset>]
```

The fields map into the arguments associated with this function. (Items enclosed in <> are integers, items in [] are optional, and items enclosed in {} indicate ''choose one of''. Note that the brackets should not appear in the actual string.) If the string is not in the Host Portable Character Encoding the result is implementation-dependent.

The *XParseGeometry* function returns a bitmask that indicates which of the four values (width, height, xoffset and yoffset) were actually found in the string and whether the x and y values are negative. By convention, −0 is not equal to +0, because the user needs to be able to say ''position the window relative to the right or bottom edge''. For each value found, the corresponding argument is updated. For each value not found, the argument is left unchanged. The bits are represented by *XValue*, *YValue*, *WidthValue*, *HeightValue*, *XNegative* or *YNegative* and are defined in **<X11/Xutil.h>**. They will be set whenever one of the values is defined or one of the signs is set.

If the function returns either the *XValue* or *YValue* flag, you should place the window at the requested position.

To construct a window's geometry information, use *XWMGeometry*.

```
int XWMGeometry(display, screen, user_geom, def_geom,
        bwidth, hints, x_return, y_return,
        width_return, height_return, gravity_return)
    Display *display;
    int screen;
    char *user_geom;
    char *def_geom;
    unsigned int bwidth;
    XSizeHints *hints;
    int *x_return, *y_return;
    int *width_return;
    int *height_return;
    int *gravity_return;
```

display   Specifies the connection to the X server.

screen	Specifies the screen.
user_geom	Specifies the user-specified geometry or NULL.
def_geom	Specifies the application's default geometry or NULL.
bwidth	Specifies the border width.
hints	Specifies the size hints for the window in its normal state.
x_return *y_return*	Return the x and y offsets.
width_return *height_return*	Return the width and height determined.
gravity_return	Returns the window gravity.

The *XWMGeometry* function combines any geometry information (given in the format used by *XParseGeometry*) specified by the user and by the calling program with size hints (usually the ones to be stored in WM_NORMAL_HINTS) and returns the position, size, and gravity (*NorthWestGravity*, *NorthEastGravity*, *SouthEastGravity* or *SouthWestGravity*) that describe the window. If the base size is not set in the *XSizeHints* structure, the minimum size is used if set. Otherwise, a base size of zero is assumed. If no minimum size is set in the hints structure, the base size is used. A mask (in the form returned by *XParseGeometry*) that describes which values came from the user specification and whether or not the position coordinates are relative to the right and bottom edges is returned. Note that these coordinates will have already been accounted for in the x_return and y_return values.

Note that invalid geometry specifications can cause a width or height of zero to be returned. The caller may pass the address of the hints win_gravity field as gravity_return to update the hints directly.

17.5   Manipulating Regions

Regions are arbitrary sets of pixel locations. Xlib provides functions for manipulating regions. The opaque type *Region* is defined in **<X11/Xutil.h>**. Xlib provides functions that you can use to manipulate regions. This section discusses how to:

- create, copy or destroy regions

- move or shrink regions

- compute with regions

- determine if regions are empty or equal

- locate a point or rectangle in a region.

17.5.1   Creating, Copying or Destroying Regions

To create a new empty region, use *XCreateRegion*.

```
Region XCreateRegion()
```

To generate a region from a polygon, use *XPolygonRegion*.

```
Region XPolygonRegion(points, n, fill_rule)
        XPoint points[];
        int n;
        int fill_rule;
```

points   Specifies an array of points.

n   Specifies the number of points in the polygon.

fill_rule   Specifies the fill-rule you want to set for the specified GC. You can pass *EvenOddRule* or *WindingRule*.

The *XPolygonRegion* function returns a region for the polygon defined by the points array. For an explanation of fill_rule, see *XCreateGC*.

To set the clip-mask of a GC to a region, use *XSetRegion*.

```
XSetRegion(display, gc, r)
        Display *display;
        GC gc;
        Region r;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

r   Specifies the region.

The *XSetRegion* function sets the clip-mask in the GC to the specified region. Once it is set in the GC, the region can be destroyed.

To deallocate the storage associated with a specified region, use *XDestroyRegion*.

```
XDestroyRegion(r)
        Region r;
```

r   Specifies the region.

17.5.2   Moving or Shrinking Regions

To move a region by a specified amount, use *XOffsetRegion*.

```
XOffsetRegion(r, dx, dy)
      Region r;
      int dx, dy;
```

r   Specifies the region.

dx

dy   Specify the x and y coordinates, which define the amount you want to move
                       the specified region.

To reduce a region by a specified amount, use *XShrinkRegion*.

```
XShrinkRegion(r, dx, dy)
      Region r;
      int dx, dy;
```

r   Specifies the region.

dx

dy   Specify the x and y coordinates, which define the amount you want to shrink
                       the specified region.

Positive values shrink the size of the region, and negative values expand the region.

17.5.3   Computing with Regions

To generate the smallest rectangle enclosing a region, use *XClipBox*.

```
XClipBox(r, rect_return)
      Region r;
      XRectangle *rect_return;
```

r   Specifies the region.

rect_return   Returns the smallest enclosing rectangle.

The *XClipBox* function returns the smallest rectangle enclosing the specified region.

To compute the intersection of two regions, use *XIntersectRegion*.

```
XIntersectRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;
```

sra

srb   Specify the two regions with which you want to perform the computation.

dr_return   Returns the result of the computation.

To compute the union of two regions, use *XUnionRegion*.

```
XUnionRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;
```

sra

srb   Specify the two regions with which you want to perform the computation.

dr_return   Returns the result of the computation.

To create a union of a source region and a rectangle, use *XUnionRectWithRegion*.

```
XUnionRectWithRegion(rectangle, src_region, dest_region_return)
      XRectangle *rectangle;
      Region src_region;
      Region dest_region_return;
```

rectangle   Specifies the rectangle.

src_region   Specifies the source region to be used.

dest_region_return Returns the destination region.

The *XUnionRectWithRegion* function updates the destination region from a union of the specified rectangle and the specified source region.

To subtract two regions, use *XSubtractRegion*.

```
XSubtractRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;
```

sra
srb   Specify the two regions with which you want to perform the computation.

dr_return   Returns the result of the computation.

The *XSubtractRegion* function subtracts srb from sra and stores the results in dr_return.

To calculate the difference between the union and intersection of two regions, use *XXorRegion*.

```
XXorRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;
```

sra
srb   Specify the two regions with which you want to perform the computation.

dr_return   Returns the result of the computation.

17.5.4   Determining if Regions Are Empty or Equal

To determine if the specified region is empty, use *XEmptyRegion*.

```
Bool XEmptyRegion(r)
      Region r;
```

r   Specifies the region.

The *XEmptyRegion* function returns *True* if the region is empty.

To determine if two regions have the same offset, size and shape, use *XEqualRegion*.

```
Bool XEqualRegion(r1, r2)
      Region r1, r2;
```

r1
r2   Specify the two regions.

The *XEqualRegion* function returns *True* if the two regions have the same offset, size and shape.

17.5.5   Locating a Point or a Rectangle in a Region

To determine if a specified point resides in a specified region, use *XPointInRegion*.

```
Bool XPointInRegion(r, x, y)
      Region r;
      int x, y;
```

r   Specifies the region.

x

y   Specify the x and y coordinates, which define the point.

The *XPointInRegion* function returns *True* if the point (x, y) is contained in the region r.

To determine if a specified rectangle is inside a region, use *XRectInRegion*.

```
int XRectInRegion(r, x, y, width, height)
      Region r;
      int x, y;
      unsigned int width, height;
```

r   Specifies the region.

x

y   Specify the x and y coordinates, which define the coordinates of the upper-left
                          corner of the rectangle.

width

height   Specify the width and height, which define the rectangle.

The *XRectInRegion* function returns *RectangleIn* if the rectangle is entirely in the specified region, *RectangleOut* if the rectangle is entirely out of the specified region, and *RectanglePart* if the rectangle is partially in the specified region.

17.6   Using Cut Buffers

Xlib provides functions to manipulate cut buffers, a very simple form of ''cut and paste'' inter-client communication. Selections are a much more powerful and useful mechanism for interchanging data between clients (see Section 5.5), and generally should be used instead of cut buffers.

Cut buffers are implemented as properties on the first root window of the display. The buffers can only contain text, in the STRING encoding. The text encoding is not changed by Xlib, when fetching or storing. Eight buffers are provided and can be accessed as a ring or as explicit buffers (numbered 0 through 7).

To store data in cut buffer 0, use *XStoreBytes*.

```
XStoreBytes(display, bytes, nbytes)
      Display *display;
      char *bytes;
      int nbytes;
```

display   Specifies the connection to the X server.

bytes   Specifies the bytes, which are not necessarily ASCII or null-terminated.

nbytes   Specifies the number of bytes to be stored.

Note that the data can have embedded null characters, and need not be null terminated. The cut buffer's contents can be retrieved later by any client calling *XFetchBytes*.

XStoreBytes can generate a *BadAlloc* error.

To store data in a specified cut buffer, use *XStoreBuffer*.

```
XStoreBuffer(display, bytes, nbytes, buffer)
      Display *display;
      char *bytes;
      int nbytes;
      int buffer;
```

display   Specifies the connection to the X server.

bytes   Specifies the bytes, which are not necessarily ASCII or null-terminated.

nbytes   Specifies the number of bytes to be stored.

buffer   Specifies the buffer in which you want to store the bytes.

If an invalid buffer is specified, the call has no effect. Note that the data can have embedded null characters, and need not be null terminated.

XStoreBuffer can generate a *BadAlloc* error.

To return data from cut buffer 0, use *XFetchBytes*.

```
char *XFetchBytes(display, nbytes_return)
      Display *display;
      int *nbytes_return;
```

display   Specifies the connection to the X server.

nbytes_return   Returns the number of bytes in the buffer.

The *XFetchBytes* function returns the number of bytes in the nbytes_return argument, if the buffer contains data. Otherwise, the function returns NULL and sets nbytes to 0. The

appropriate amount of storage is allocated and the pointer returned. The client must free this storage when finished with it by calling *XFree*.

To return data from a specified cut buffer, use *XFetchBuffer*.

```
char *XFetchBuffer(display, nbytes_return, buffer)
      Display *display;
      int *nbytes_return;
      int buffer;
```

display   Specifies the connection to the X server.

nbytes_return   Returns the number of bytes in the buffer.

buffer   Specifies the buffer from which you want the stored data returned.

The *XFetchBuffer* function returns zero to the nbytes_return argument if there is no data in the buffer or if an invalid buffer is specified.

To rotate the cut buffers, use *XRotateBuffers*.

```
XRotateBuffers(display, rotate)
      Display *display;
      int rotate;
```

display   Specifies the connection to the X server.

rotate   Specifies how much to rotate the cut buffers.

The *XRotateBuffers* function rotates the cut buffers, such that buffer 0 becomes buffer n, buffer 1 becomes n + 1 mod 8, and so on. This cut buffer numbering is global to the display. Note that *XRotateBuffers* generates *BadMatch* errors if any of the eight buffers have not been created.

17.7   **Determining the Appropriate Visual Type**

A single display can support multiple screens. Each screen can have several different visual types supported at different depths. You can use the functions described in this section to determine which visual to use for your application.

The functions in this section use the visual information masks and the *XVisualInfo* structure, which is defined in **<X11/Xutil.h>** and contains:

```
/* Visual information mask bits */

#define    VisualNoMask              0x0
#define    VisualIDMask              0x1
#define    VisualScreenMask          0x2
#define    VisualDepthMask           0x4
#define    VisualClassMask           0x8
#define    VisualRedMaskMask         0x10
#define    VisualGreenMaskMask       0x20
#define    VisualBlueMaskMask        0x40
#define    VisualColormapSizeMask    0x80
#define    VisualBitsPerRGBMask      0x100
#define    VisualAllMask             0x1FF

/* Values */

typedef struct {
       Visual *visual;
       VisualID visualid;
       int screen;
       unsigned int depth;
       int class;
       unsigned long red_mask;
       unsigned long green_mask;
       unsigned long blue_mask;
       int colormap_size;
       int bits_per_rgb;
} XVisualInfo;
```

To obtain a list of visual information structures that match a specified template, use *XGetVisualInfo*.

```
XVisualInfo *XGetVisualInfo(display, vinfo_mask, vinfo_template,
        nitems_return)
       Display *display;
       long vinfo_mask;
       XVisualInfo *vinfo_template;
       int *nitems_return;
```

display   Specifies the connection to the X server.

vinfo_mask   Specifies the visual mask value.

vinfo_template   Specifies the visual attributes that are to be used in matching the visual structures.

nitems_return   Returns the number of matching visual structures.

The *XGetVisualInfo* function returns a list of visual structures that have attributes equal to the attributes specified by vinfo_template. If no visual structures match the template using the specified vinfo_mask, *XGetVisualInfo* returns a NULL. To free the data returned by this function, use *XFree*.

To obtain the visual information that matches the specified depth and class of the screen, use *XMatchVisualInfo*.

```
Status XMatchVisualInfo(display, screen, depth, class,
        vinfo_return)
     Display *display;
     int screen;
     int depth;
     int class;
     XVisualInfo *vinfo_return;
```

display   Specifies the connection to the X server.

screen   Specifies the screen.

depth   Specifies the depth of the screen.

class   Specifies the class of the screen.

vinfo_return   Returns the matched visual information.

The *XMatchVisualInfo* function returns the visual information for a visual that matches the specified depth and class for a screen. Because multiple visuals that match the specified depth and class can exist, the exact visual chosen is undefined. If a visual is found, *XMatchVisualInfo* returns nonzero and the information on the visual to vinfo_return. Otherwise, when a visual is not found, *XMatchVisualInfo* returns zero.

17.8   Manipulating Images

Xlib provides several functions that perform basic operations on images. All operations on images are defined using an *XImage* structure, as defined in **<X11/Xlib.h>**. Because the number of different types of image formats can be very large, this hides details of image storage properly from applications.

This section describes the functions for generic operations on images. Manufacturers can provide very fast implementations of these for the formats frequently encountered on their hardware. These functions are neither sufficient nor desirable to use for general image processing. Rather, they are here to provide minimal functions on screen format images. The basic operations for getting and putting images are *XGetImage* and *XPutImage*.

Note that no functions have been defined, as yet, to read and write images to and from disk files.

The *XImage* structure describes an image as it exists in the client's memory. The user can request that some of the members such as height, width, and xoffset be changed when the image is sent to the server. Note that bytes_per_line in concert with offset can be used to extract a subset of the image. Other members (for example, byte order, bitmap_unit, and so on) are characteristics of both the image and the server. If these members differ between the image and the server, *XPutImage* makes the appropriate conversions. The first byte of the first line of plane n must be located at the address (data + (n * height * bytes_per_line)). For a description of the *XImage* structure, see Section 9.7.

To allocate sufficient memory for an *XImage* structure, use *XCreateImage*.

```
XImage *XCreateImage(display, visual, depth, format,
        offset, data, width, height,
        bitmap_pad, bytes_per_line)
     Display *display;
     Visual *visual;
     unsigned int depth;
     int format;
     int offset;
     char *data;
     unsigned int width;
     unsigned int height;
     int bitmap_pad;
     int bytes_per_line;
```

display   Specifies the connection to the X server.

visual   Specifies the *Visual* structure.

depth   Specifies the depth of the image.

format   Specifies the format for the image. You can pass *XYBitmap*, *XYPixmap* or *ZPixmap*.

offset   Specifies the number of pixels to ignore at the beginning of the scanline.

data   Specifies the image data.

width   Specifies the width of the image, in pixels.

height   Specifies the height of the image, in pixels.

bitmap_pad   Specifies the quantum of a scanline (8, 16 or 32). In other words, the start of one scanline is separated in client memory from the start of the next scanline by an integer multiple of this many bits.

bytes_per_line   Specifies the number of bytes in the client image between the start of one scanline and the start of the next.

The *XCreateImage* function allocates the memory needed for an *XImage* structure for the specified display but does not allocate space for the image itself. Rather, it initializes the structure byte-order, bit-order and bitmap-unit values from the display and returns a pointer to the *XImage* structure. The red, green and blue mask values are defined for Z format images only and are derived from the *Visual* structure passed in. Other values also are passed in. The offset permits the rapid displaying of the image without requiring each scanline to be shifted into position. If you pass a zero value in bytes_per_line, Xlib assumes that the scanlines are contiguous in memory and calculates the value of bytes_per_line itself.

Note that when the image is created using *XCreateImage*, *XGetImage* or *XSubImage*, the destroy procedure that the *XDestroyImage* function calls frees both the image structure and the data pointed to by the image structure.

The basic functions used to get a pixel, set a pixel, create a subimage, and add a constant value to an image are defined in the image object. The functions in this section are really macro invocations of the functions in the image object and are defined in **<X11/Xutil.h>**.

To obtain a pixel value in an image, use *XGetPixel*.

```
unsigned long XGetPixel(ximage, x, y)
      XImage *ximage;
      int x;
      int y;
```

ximage   Specifies the image.

x

y   Specify the x and y coordinates.

The *XGetPixel* function returns the specified pixel from the named image. The pixel value is returned in normalized format (that is, the least-significant byte of the long is the least-significant byte of the pixel). The image must contain the x and y coordinates.

To set a pixel value in an image, use *XPutPixel*.

```
XPutPixel(ximage, x, y, pixel)
      XImage *ximage;
      int x;
      int y;
      unsigned long pixel;
```

ximage   Specifies the image.

x

y   Specify the x and y coordinates.

pixel   Specifies the new pixel value.

The *XPutPixel* function overwrites the pixel in the named image with the specified pixel value. The input pixel value must be in normalized format (that is, the least-significant byte of the long is the least-significant byte of the pixel). The image must contain the x and y coordinates.

To create a subimage, use *XSubImage*.

```
XImage *XSubImage(ximage, x, y, subimage_width, subimage_height)
      XImage *ximage;
      int x;
      int y;
      unsigned int subimage_width;
      unsigned int subimage_height;
```

ximage   Specifies the image.

x

y   Specify the x and y coordinates.

subimage_width   Specifies the width of the new subimage, in pixels.

subimage_height   Specifies the height of the new subimage, in pixels.

The *XSubImage* function creates a new image that is a subsection of an existing one. It allocates the memory necessary for the new *XImage* structure and returns a pointer to the new image. The data is copied from the source image, and the image must contain the rectangle defined by x, y, subimage_width and subimage_height.

To increment each pixel in an image by a constant value, use *XAddPixel*.

```
XAddPixel(ximage, value)
      XImage *ximage;
      long value;
```

ximage   Specifies the image.

value   Specifies the constant value that is to be added.

The *XAddPixel* function adds a constant value to every pixel in an image. It is useful when you have a base pixel value from allocating color resources and need to manipulate the image to that form.

To deallocate the memory allocated in a previous call to *XCreateImage*, use *XDestroyImage*.

```
XDestroyImage(ximage)
        XImage *ximage;
```

ximage   Specifies the image.

The *XDestroyImage* function deallocates the memory associated with the *XImage* structure.

Note that when the image is created using *XCreateImage*, *XGetImage* or *XSubImage*, the destroy procedure that this macro calls frees both the image structure and the data pointed to by the image structure.

17.9   Manipulating Bitmaps

Xlib provides functions that you can use to read a bitmap from a file, save a bitmap to a file, or create a bitmap. This section describes those functions that transfer bitmaps to and from the client's file system, thus allowing their reuse in a later connection (for example, from an entirely different client or to a different display or server).

The X version 11 bitmap file format is:

```
#define name_width width
#define name_height height
#define name_x_hot x
#define name_y_hot y
static unsigned char name_bits[] = { 0xNN,... }
```

The lines for the variables ending with _x_hot and _y_hot suffixes are optional because they are present only if a hotspot has been defined for this bitmap. The lines for the other variables are required. The word ''unsigned'' is optional; that is, the type of the _bits array can be char or unsigned char. The _bits array must be large enough to contain the size bitmap. The bitmap unit is eight. The name is derived from the name of the file that you specified on the original command line by deleting the directory path and extension.

To read a bitmap from a file, use *XReadBitmapFile*.

```
int XReadBitmapFile(display, d, filename, width_return,
        height_return, bitmap_return, x_hot_return,
        y_hot_return)
    Display *display;
    Drawable d;
    char *filename;
    unsigned int *width_return, *height_return;
    Pixmap *bitmap_return;
    int *x_hot_return, *y_hot_return;
```

display	Specifies the connection to the X server.
d	Specifies the drawable that indicates the screen.
filename	Specifies the file name to use. The format of the file name is operating system-dependent.
width_return *height_return*	Return the width and height values of the read in bitmap file.
bitmap_return	Returns the bitmap that is created.
x_hot_return *y_hot_return*	Return the hotspot coordinates.

The *XReadBitmapFile* function reads in a file containing a bitmap. The file is parsed in the encoding of the current locale. The ability to read other than the standard format is implementation-dependent. If the file cannot be opened, *XReadBitmapFile* returns *BitmapOpenFailed*. If the file can be opened but does not contain valid bitmap data, it returns *BitmapFileInvalid*. If insufficient working storage is allocated, it returns *BitmapNoMemory*. If the file is readable and valid, it returns *BitmapSuccess*.

XReadBitmapFile returns the bitmap's height and width, as read from the file, to width_return and height_return. It then creates a pixmap of the appropriate size, reads the bitmap data from the file into the pixmap, and assigns the pixmap to the caller's variable bitmap. The caller must

free the bitmap using *XFreePixmap* when finished. If *name*_x_hot and *name*_y_hot exist, *XReadBitmapFile* returns them to x_hot_return and y_hot_return; otherwise, it returns −1,−1.

XReadBitmapFile can generate *BadAlloc* and *BadDrawable* errors.

To write out a bitmap to a file, use *XWriteBitmapFile*.

```
int XWriteBitmapFile(display, filename, bitmap, width,
        height, x_hot, y_hot)
    Display *display;
    char *filename;
    Pixmap bitmap;
    unsigned int width, height;
    int x_hot, y_hot;
```

display   Specifies the connection to the X server.

filename   Specifies the file name to use. The format of the file name is operating system-dependent.

bitmap   Specifies the bitmap.

width
height   Specify the width and height.

x_hot
y_hot   Specify where to place the hotspot coordinates (or −1,−1 if none are present) in the file.

The *XWriteBitmapFile* function writes a bitmap out to a file in the X version 11 format. The file is written in the encoding of the current locale. If the file cannot be opened for writing, it returns *BitmapOpenFailed*. If insufficient memory is allocated, *XWriteBitmapFile* returns *BitmapNoMemory*; otherwise, on no error, it returns *BitmapSuccess*. If x_hot and y_hot are not −1, −1, *XWriteBitmapFile* writes them out as the hotspot coordinates for the bitmap.

XWriteBitmapFile can generate *BadDrawable* and *BadMatch* errors.

To create a pixmap and then store bitmap-format data into it, use *XCreatePixmapFromBitmapData*.

```
Pixmap XCreatePixmapFromBitmapData(display, d, data,
        width, height, fg, bg, depth)
    Display *display;
    Drawable d;
    char *data;
    unsigned int width, height;
    unsigned long fg, bg;
    unsigned int depth;
```

display   Specifies the connection to the X server.

d   Specifies the drawable that indicates the screen.

data   Specifies the data in bitmap format.

width
height   Specify the width and height.

fg
bg   Specify the foreground and background pixel values to use.

depth   Specifies the depth of the pixmap.

The *XCreatePixmapFromBitmapData* function creates a pixmap of the given depth and then does a bitmap-format *XPutImage* of the data into it. The depth must be supported by the screen of the specified drawable, or a *BadMatch* error results.

XCreatePixmapFromBitmapData can generate *BadAlloc* and *BadMatch* errors.

To include a bitmap written out by *XWriteBitmapFile* in a program directly, as opposed to reading it in every time at run time, use *XCreateBitmapFromData*.

```
Pixmap XCreateBitmapFromData(display, d, data, width,
        height)
      Display *display;
      Drawable d;
      char *data;
      unsigned int width, height;
```

display   Specifies the connection to the X server.

d   Specifies the drawable that indicates the screen.

data   Specifies the location of the bitmap data.

width
height   Specify the width and height.

The *XCreateBitmapFromData* function allows you to include in your C program (using #include) a bitmap file that was written out by *XWriteBitmapFile* (X version 11 format only) without reading in the bitmap file. The following example creates a gray bitmap:

```
#include "gray.bitmap"

Pixmap bitmap;
bitmap = XCreateBitmapFromData(display, window, gray_bits, gray_width,
        gray_height);
```

If insufficient working storage was allocated, *XCreateBitmapFromData* returns *None*. It is your responsibility to free the bitmap using *XFreePixmap* when finished.

XCreateBitmapFromData can generate a *BadAlloc* error.

17.10   Using the Context Manager

The context manager provides a way of associating data with an X resource ID (mostly typically a window) in your program. Note that this is local to your program; the data is not stored in the server on a property list. Any amount of data in any number of pieces can be associated with a resource ID, and each piece of data has a type associated with it. The context manager requires knowledge of the resource ID and type to store or retrieve data.

Essentially, the context manager can be viewed as a two-dimensional, sparse array: one dimension is subscripted by the X resource ID and the other by a context type field. Each entry in the array contains a pointer to the data. Xlib provides context management functions with which you can save data values, get data values, delete entries and create a unique context type. The symbols used are in <**X11/Xutil.h**>.

To save a data value that corresponds to a resource ID and context type, use *XSaveContext*.

```
int XSaveContext(display, rid, context, data)
      Display *display;
      XID rid;
      XContext context;
      XPointer data;
```

display   Specifies the connection to the X server.

rid   Specifies the resource ID with which the data is associated.

context   Specifies the context type to which the data belongs.

data   Specifies the data to be associated with the window and type.

If an entry with the specified resource ID and type already exists, *XSaveContext* overrides it with the specified context. The *XSaveContext* function returns a nonzero error code if an error has occurred and zero otherwise. Possible errors are *XCNOMEM* (out of memory).

To get the data associated with a resource ID and type, use *XFindContext*.

```
int XFindContext(display, rid, context, data_return)
      Display *display;
      XID rid;
      XContext context;
      XPointer *data_return;
```

display   Specifies the connection to the X server.

rid   Specifies the resource ID with which the data is associated.

context   Specifies the context type to which the data belongs.

data_return   Returns the data.

Because it is a return value, the data is a pointer. The *XFindContext* function returns a nonzero error code if an error has occurred and zero otherwise. Possible errors are *XCNOENT* (context-not-found).

To delete an entry for a given resource ID and type, use *XDeleteContext*.

```
int XDeleteContext(display, rid, context)
      Display *display;
      XID rid;
      XContext context;
```

display   Specifies the connection to the X server.

rid   Specifies the resource ID with which the data is associated.

context   Specifies the context type to which the data belongs.

The *XDeleteContext* function deletes the entry for the given resource ID and type from the data structure. This function returns the same error codes that *XFindContext* returns if called with the same arguments. *XDeleteContext* does not free the data whose address was saved.

To create a unique context type that may be used in subsequent calls to *XSaveContext* and *XFindContext*, use *XUniqueContext*.

```
XContext XUniqueContext()
```

Additional X/Open Requirements

18.1 The ''cursor'' Font

All X/Open-compliant servers will provide at least the ''cursor'' font described in this section. This font will also have a full XLFD name, which is implementation-defined. The ''cursor'' font will contain cursor bitmaps, and will be the font used by the *XCreateFontCursor*() function. This font will include at least the cursors specified in the following table. The data bitmap for a cursor will be the character with the code given under ''Position'' in the table; the mask bitmap will be the character with the code 1 greater.

The header file **<X11/cursorfont.h>** will define at least the symbols listed in the table, with the values given under ''Position''. In addition, it will define values for any other bitmaps in the font; such values will be even and between 0 and 254 inclusive, and the symbols will all begin with *XC_*. The header file will also define the symbol *XC_num_glyphs* with a value 2 greater than the largest cursor name symbol defined (and so at least 150).

The glyphs shown are examples; it is not necessary to match them exactly.

Group	Defined Symbol	Position	Glyph
Default cursor	XC_X_cursor	0	✖
Corner cursors	XC_bottom_left_corner	12	⌐
	XC_bottom_right_corner	14	⌐
	XC_top_left_corner	134	⌐
	XC_top_right_corner	136	⌐
Arrow cursors	XC_arrow	2	➚
	XC_based_arrow_down	4	⊤
	XC_based_arrow_up	6	⊥
	XC_double_arrow	42	⇕
	XC_question_arrow	92	?
	XC_sb_h_double_arrow	108	↔
	XC_sb_v_double_arrow	116	↕
	XC_top_left_arrow	132	↖
Tee cursors	XC_bottom_tee	18	⊥
	XC_left_tee	72	⊢
	XC_right_tee	98	⊣
	XC_top_tee	140	⊤
Side cursors	XC_bottom_side	16	↓
	XC_left_side	70	←

Group	Defined Symbol	Position	Glyph
	XC_right_side	96	→|
	XC_top_side	138	↑
Angle cursors	XC_ll_angle	76	└
	XC_lr_angle	78	┘
	XC_ul_angle	144	┌
	XC_ur_angle	148	┐
Circle cursors	XC_circle	24	O
Cross cursors	XC_cross	30	╫
	XC_crosshair	34	+
Pointer cursors	XC_center_ptr	22	↑
	XC_left_ptr	68	↖
	XC_right_ptr	94	↗

18.2   KEYSYM Values

The header file **<X11/keysymdef.h>** shall include at least definitions for the keysyms listed in this section. Most keysyms in this file are only defined if a ''feature test macro'' has been defined (by **#define**) before this file is included. The feature test macros, and the codes used to indicate them in the main list, are shown in the following table.

Macro	Code
XK_LATIN1	1
XK_LATIN2	2
XK_LATIN3	3
XK_LATIN4	4
XK_ARABIC	A
XK_CYRILLIC	C
XK_GREEK	G
XK_HEBREW	H
XK_KATAKANA	K
XK_APL	L
XK_MISCELLANY	M
XK_PUBLISHING	P
XK_SPECIAL	S
XK_TECHNICAL	T

Including the header file **<X11/keysym.h>** shall be equivalent to defining the symbols *XK_LATIN1*, *XK_LATIN2*, *XK_LATIN3*, *XK_LATIN4*, *XK_GREEK* and *XK_MISCELLANY*, and then including the header file **<X11/keysymdef.h>**.

The full list of keysyms is given in the following table. ''Code'' is the code for the feature test macro (if any) required to be defined for this keysym to be defined. ''Value'' is the value of the keysym; this is in hexadecimal, even though no prefix is given. ''Name'' is the name of the defined symbol; a value may have several names, and these are listed on separate lines. Where a name is suffixed with †, this name will be defined, but should not be used in new software.

Code	Value	Name	Code	Value	Name
1	20	XK_space	1	2F	XK_slash
1	21	XK_exclam	1	30	XK_0
1	22	XK_quotedbl	1	31	XK_1
1	23	XK_numbersign	1	32	XK_2
1	24	XK_dollar	1	33	XK_3
1	25	XK_percent	1	34	XK_4
1	26	XK_ampersand	1	35	XK_5
1	27	XK_apostrophe	1	36	XK_6
1	27	XK_quoteright †	1	37	XK_7
1	28	XK_parenleft	1	38	XK_8
1	29	XK_parenright	1	39	XK_9
1	2A	XK_asterisk	1	3A	XK_colon
1	2B	XK_plus	1	3B	XK_semicolon
1	2C	XK_comma	1	3C	XK_less
1	2D	XK_minus	1	3D	XK_equal
1	2E	XK_period	1	3E	XK_greater

Code	Value	Name
1	3F	XK_question
1	40	XK_at
1	41	XK_A
1	42	XK_B
1	43	XK_C
1	44	XK_D
1	45	XK_E
1	46	XK_F
1	47	XK_G
1	48	XK_H
1	49	XK_I
1	4A	XK_J
1	4B	XK_K
1	4C	XK_L
1	4D	XK_M
1	4E	XK_N
1	4F	XK_O
1	50	XK_P
1	51	XK_Q
1	52	XK_R
1	53	XK_S
1	54	XK_T
1	55	XK_U
1	56	XK_V
1	57	XK_W
1	58	XK_X
1	59	XK_Y
1	5A	XK_Z
1	5B	XK_bracketleft
1	5C	XK_backslash
1	5D	XK_bracketright
1	5E	XK_asciicircum
1	5F	XK_underscore
1	60	XK_grave
1	60	XK_quoteleft †
1	61	XK_a
1	62	XK_b
1	63	XK_c
1	64	XK_d
1	65	XK_e
1	66	XK_f
1	67	XK_g
1	68	XK_h
1	69	XK_i
1	6A	XK_j
1	6B	XK_k
1	6C	XK_l
1	6D	XK_m
1	6E	XK_n
1	6F	XK_o
1	70	XK_p
1	71	XK_q
1	72	XK_r

Code	Value	Name
1	73	XK_s
1	74	XK_t
1	75	XK_u
1	76	XK_v
1	77	XK_w
1	78	XK_x
1	79	XK_y
1	7A	XK_z
1	7B	XK_braceleft
1	7C	XK_bar
1	7D	XK_braceright
1	7E	XK_asciitilde
1	A0	XK_nobreakspace
1	A1	XK_exclamdown
1	A2	XK_cent
1	A3	XK_sterling
1	A4	XK_currency
1	A5	XK_yen
1	A6	XK_brokenbar
1	A7	XK_section
1	A8	XK_diaeresis
1	A9	XK_copyright
1	AA	XK_ordfeminine
1	AB	XK_guillemotleft
1	AC	XK_notsign
1	AD	XK_hyphen
1	AE	XK_registered
1	AF	XK_macron
1	B0	XK_degree
1	B1	XK_plusminus
1	B2	XK_twosuperior
1	B3	XK_threesuperior
1	B4	XK_acute
1	B5	XK_mu
1	B6	XK_paragraph
1	B7	XK_periodcentered
1	B8	XK_cedilla
1	B9	XK_onesuperior
1	BA	XK_masculine
1	BB	XK_guillemotright
1	BC	XK_onequarter
1	BD	XK_onehalf
1	BE	XK_threequarters
1	BF	XK_questiondown
1	C0	XK_Agrave
1	C1	XK_Aacute
1	C2	XK_Acircumflex
1	C3	XK_Atilde
1	C4	XK_Adiaeresis
1	C5	XK_Aring
1	C6	XK_AE
1	C7	XK_Ccedilla
1	C8	XK_Egrave

Code	Value	Name
1	C9	XK_Eacute
1	CA	XK_Ecircumflex
1	CB	XK_Ediaeresis
1	CC	XK_Igrave
1	CD	XK_Iacute
1	CE	XK_Icircumflex
1	CF	XK_Idiaeresis
1	D0	XK_ETH
1	D0	XK_Eth †
1	D1	XK_Ntilde
1	D2	XK_Ograve
1	D3	XK_Oacute
1	D4	XK_Ocircumflex
1	D5	XK_Otilde
1	D6	XK_Odiaeresis
1	D7	XK_multiply
1	D8	XK_Ooblique
1	D9	XK_Ugrave
1	DA	XK_Uacute
1	DB	XK_Ucircumflex
1	DC	XK_Udiaeresis
1	DD	XK_Yacute
1	DE	XK_THORN
1	DE	XK_Thorn †
1	DF	XK_ssharp
1	E0	XK_agrave
1	E1	XK_aacute
1	E2	XK_acircumflex
1	E3	XK_atilde
1	E4	XK_adiaeresis
1	E5	XK_aring
1	E6	XK_ae
1	E7	XK_ccedilla
1	E8	XK_egrave
1	E9	XK_eacute
1	EA	XK_ecircumflex
1	EB	XK_ediaeresis
1	EC	XK_igrave
1	ED	XK_iacute
1	EE	XK_icircumflex
1	EF	XK_idiaeresis
1	F0	XK_eth
1	F1	XK_ntilde
1	F2	XK_ograve
1	F3	XK_oacute
1	F4	XK_ocircumflex
1	F5	XK_otilde
1	F6	XK_odiaeresis
1	F7	XK_division
1	F8	XK_oslash
1	F9	XK_ugrave
1	FA	XK_uacute
1	FB	XK_ucircumflex

Code	Value	Name
1	FC	XK_udiaeresis
1	FD	XK_yacute
1	FE	XK_thorn
1	FF	XK_ydiaeresis
2	1A1	XK_Aogonek
2	1A2	XK_breve
2	1A3	XK_Lstroke
2	1A5	XK_Lcaron
2	1A6	XK_Sacute
2	1A9	XK_Scaron
2	1AA	XK_Scedilla
2	1AB	XK_Tcaron
2	1AC	XK_Zacute
2	1AE	XK_Zcaron
2	1AF	XK_Zabovedot
2	1B1	XK_aogonek
2	1B2	XK_ogonek
2	1B3	XK_lstroke
2	1B5	XK_lcaron
2	1B6	XK_sacute
2	1B7	XK_caron
2	1B9	XK_scaron
2	1BA	XK_scedilla
2	1BB	XK_tcaron
2	1BC	XK_zacute
2	1BD	XK_doubleacute
2	1BE	XK_zcaron
2	1BF	XK_zabovedot
2	1C0	XK_Racute
2	1C3	XK_Abreve
2	1C5	XK_Lacute
2	1C6	XK_Cacute
2	1C8	XK_Ccaron
2	1CA	XK_Eogonek
2	1CC	XK_Ecaron
2	1CF	XK_Dcaron
2	1D0	XK_Dstroke
2	1D1	XK_Nacute
2	1D2	XK_Ncaron
2	1D5	XK_Odoubleacute
2	1D8	XK_Rcaron
2	1D9	XK_Uring
2	1DB	XK_Udoubleacute
2	1DE	XK_Tcedilla
2	1E0	XK_racute
2	1E3	XK_abreve
2	1E5	XK_lacute
2	1E6	XK_cacute
2	1E8	XK_ccaron
2	1EA	XK_eogonek
2	1EC	XK_ecaron
2	1EF	XK_dcaron
2	1F0	XK_dstroke

Code	Value	Name
2	1F1	XK_nacute
2	1F2	XK_ncaron
2	1F5	XK_odoubleacute
2	1F8	XK_rcaron
2	1F9	XK_uring
2	1FB	XK_udoubleacute
2	1FE	XK_tcedilla
2	1FF	XK_abovedot
3	2A1	XK_Hstroke
3	2A6	XK_Hcircumflex
3	2A9	XK_Iabovedot
3	2AB	XK_Gbreve
3	2AC	XK_Jcircumflex
3	2B1	XK_hstroke
3	2B6	XK_hcircumflex
3	2B9	XK_idotless
3	2BB	XK_gbreve
3	2BC	XK_jcircumflex
3	2C5	XK_Cabovedot
3	2C6	XK_Ccircumflex
3	2D5	XK_Gabovedot
3	2D8	XK_Gcircumflex
3	2DD	XK_Ubreve
3	2DE	XK_Scircumflex
3	2E5	XK_cabovedot
3	2E6	XK_ccircumflex
3	2F5	XK_gabovedot
3	2F8	XK_gcircumflex
3	2FD	XK_ubreve
3	2FE	XK_scircumflex
4	3A2	XK_kra
4	3A2	XK_kappa †
4	3A3	XK_Rcedilla
4	3A5	XK_Itilde
4	3A6	XK_Lcedilla
4	3AA	XK_Emacron
4	3AB	XK_Gcedilla
4	3AC	XK_Tslash
4	3B3	XK_rcedilla
4	3B5	XK_itilde
4	3B6	XK_lcedilla
4	3BA	XK_emacron
4	3BB	XK_gcedilla
4	3BC	XK_tslash
4	3BD	XK_ENG
4	3BF	XK_eng
4	3C0	XK_Amacron
4	3C7	XK_Iogonek
4	3CC	XK_Eabovedot
4	3CF	XK_Imacron
4	3D1	XK_Ncedilla
4	3D2	XK_Omacron
4	3D3	XK_Kcedilla

Code	Value	Name
4	3D9	XK_Uogonek
4	3DD	XK_Utilde
4	3DE	XK_Umacron
4	3E0	XK_amacron
4	3E7	XK_iogonek
4	3EC	XK_eabovedot
4	3EF	XK_imacron
4	3F1	XK_ncedilla
4	3F2	XK_omacron
4	3F3	XK_kcedilla
4	3F9	XK_uogonek
4	3FD	XK_utilde
4	3FE	XK_umacron
K	47E	XK_overline
K	4A1	XK_kana_fullstop
K	4A2	XK_kana_openingbracket
K	4A3	XK_kana_closingbracket
K	4A4	XK_kana_comma
K	4A5	XK_kana_conjunctive
K	4A5	XK_kana_middledot †
K	4A6	XK_kana_WO
K	4A7	XK_kana_a
K	4A8	XK_kana_i
K	4A9	XK_kana_u
K	4AA	XK_kana_e
K	4AB	XK_kana_o
K	4AC	XK_kana_ya
K	4AD	XK_kana_yu
K	4AE	XK_kana_yo
K	4AF	XK_kana_tsu
K	4AF	XK_kana_tu †
K	4B0	XK_prolongedsound
K	4B1	XK_kana_A
K	4B2	XK_kana_I
K	4B3	XK_kana_U
K	4B4	XK_kana_E
K	4B5	XK_kana_O
K	4B6	XK_kana_KA
K	4B7	XK_kana_KI
K	4B8	XK_kana_KU
K	4B9	XK_kana_KE
K	4BA	XK_kana_KO
K	4BB	XK_kana_SA
K	4BC	XK_kana_SHI
K	4BD	XK_kana_SU
K	4BE	XK_kana_SE
K	4BF	XK_kana_SO
K	4C0	XK_kana_TA
K	4C1	XK_kana_CHI
K	4C1	XK_kana_TI †
K	4C2	XK_kana_TSU
K	4C2	XK_kana_TU †
K	4C3	XK_kana_TE

Code	Value	Name
K	4C4	XK_kana_TO
K	4C5	XK_kana_NA
K	4C6	XK_kana_NI
K	4C7	XK_kana_NU
K	4C8	XK_kana_NE
K	4C9	XK_kana_NO
K	4CA	XK_kana_HA
K	4CB	XK_kana_HI
K	4CC	XK_kana_FU
K	4CC	XK_kana_HU †
K	4CD	XK_kana_HE
K	4CE	XK_kana_HO
K	4CF	XK_kana_MA
K	4D0	XK_kana_MI
K	4D1	XK_kana_MU
K	4D2	XK_kana_ME
K	4D3	XK_kana_MO
K	4D4	XK_kana_YA
K	4D5	XK_kana_YU
K	4D6	XK_kana_YO
K	4D7	XK_kana_RA
K	4D8	XK_kana_RI
K	4D9	XK_kana_RU
K	4DA	XK_kana_RE
K	4DB	XK_kana_RO
K	4DC	XK_kana_WA
K	4DD	XK_kana_N
K	4DE	XK_voicedsound
K	4DF	XK_semivoicedsound
A	5AC	XK_Arabic_comma
A	5BB	XK_Arabic_semicolon
A	5BF	XK_Arabic_question_mark
A	5C1	XK_Arabic_hamza
A	5C2	XK_Arabic_maddaonalef
A	5C3	XK_Arabic_hamzaonalef
A	5C4	XK_Arabic_hamzaonwaw
A	5C5	XK_Arabic_hamzaunderalef
A	5C6	XK_Arabic_hamzaonyeh
A	5C7	XK_Arabic_alef
A	5C8	XK_Arabic_beh
A	5C9	XK_Arabic_tehmarbuta
A	5CA	XK_Arabic_teh
A	5CB	XK_Arabic_theh
A	5CC	XK_Arabic_jeem
A	5CD	XK_Arabic_hah
A	5CE	XK_Arabic_khah
A	5CF	XK_Arabic_dal
A	5D0	XK_Arabic_thal
A	5D1	XK_Arabic_ra
A	5D2	XK_Arabic_zain
A	5D3	XK_Arabic_seen
A	5D4	XK_Arabic_sheen
A	5D5	XK_Arabic_sad

Code	Value	Name
A	5D6	XK_Arabic_dad
A	5D7	XK_Arabic_tah
A	5D8	XK_Arabic_zah
A	5D9	XK_Arabic_ain
A	5DA	XK_Arabic_ghain
A	5E0	XK_Arabic_tatweel
A	5E1	XK_Arabic_feh
A	5E2	XK_Arabic_qaf
A	5E3	XK_Arabic_kaf
A	5E4	XK_Arabic_lam
A	5E5	XK_Arabic_meem
A	5E6	XK_Arabic_noon
A	5E7	XK_Arabic_ha
A	5E7	XK_Arabic_heh †
A	5E8	XK_Arabic_waw
A	5E9	XK_Arabic_alefmaksura
A	5EA	XK_Arabic_yeh
A	5EB	XK_Arabic_fathatan
A	5EC	XK_Arabic_dammatan
A	5ED	XK_Arabic_kasratan
A	5EE	XK_Arabic_fatha
A	5EF	XK_Arabic_damma
A	5F0	XK_Arabic_kasra
A	5F1	XK_Arabic_shadda
A	5F2	XK_Arabic_sukun
C	6A1	XK_Serbian_dje
C	6A2	XK_Macedonia_gje
C	6A3	XK_Cyrillic_io
C	6A4	XK_Ukrainian_ie
C	6A4	XK_Ukranian_je †
C	6A5	XK_Macedonia_dse
C	6A6	XK_Ukrainian_i
C	6A6	XK_Ukranian_i †
C	6A7	XK_Ukrainian_yi
C	6A7	XK_Ukranian_yi †
C	6A8	XK_Cyrillic_je
C	6A8	XK_Serbian_je †
C	6A9	XK_Cyrillic_lje
C	6A9	XK_Serbian_lje †
C	6AA	XK_Cyrillic_nje
C	6AA	XK_Serbian_nje †
C	6AB	XK_Serbian_tshe
C	6AC	XK_Macedonia_kje
C	6AE	XK_Byelorussian_shortu
C	6AF	XK_Cyrillic_dzhe
C	6AF	XK_Serbian_dze †
C	6B0	XK_numerosign
C	6B1	XK_Serbian_DJE
C	6B2	XK_Macedonia_GJE
C	6B3	XK_Cyrillic_IO
C	6B4	XK_Ukrainian_IE
C	6B4	XK_Ukranian_JE †
C	6B5	XK_Macedonia_DSE

Code	Value	Name
C	6B6	XK_Ukrainian_I
C	6B6	XK_Ukranian_I †
C	6B7	XK_Ukrainian_YI
C	6B7	XK_Ukranian_YI †
C	6B8	XK_Cyrillic_JE
C	6B8	XK_Serbian_JE †
C	6B9	XK_Cyrillic_LJE
C	6B9	XK_Serbian_LJE †
C	6BA	XK_Cyrillic_NJE
C	6BA	XK_Serbian_NJE †
C	6BB	XK_Serbian_TSHE
C	6BC	XK_Macedonia_KJE
C	6BE	XK_Byelorussian_SHORTU
C	6BF	XK_Cyrillic_DZHE
C	6BF	XK_Serbian_DZE †
C	6C0	XK_Cyrillic_yu
C	6C1	XK_Cyrillic_a
C	6C2	XK_Cyrillic_be
C	6C3	XK_Cyrillic_tse
C	6C4	XK_Cyrillic_de
C	6C5	XK_Cyrillic_ie
C	6C6	XK_Cyrillic_ef
C	6C7	XK_Cyrillic_ghe
C	6C8	XK_Cyrillic_ha
C	6C9	XK_Cyrillic_i
C	6CA	XK_Cyrillic_shorti
C	6CB	XK_Cyrillic_ka
C	6CC	XK_Cyrillic_el
C	6CD	XK_Cyrillic_em
C	6CE	XK_Cyrillic_en
C	6CF	XK_Cyrillic_o
C	6D0	XK_Cyrillic_pe
C	6D1	XK_Cyrillic_ya
C	6D2	XK_Cyrillic_er
C	6D3	XK_Cyrillic_es
C	6D4	XK_Cyrillic_te
C	6D5	XK_Cyrillic_u
C	6D6	XK_Cyrillic_zhe
C	6D7	XK_Cyrillic_ve
C	6D8	XK_Cyrillic_softsign
C	6D9	XK_Cyrillic_yeru
C	6DA	XK_Cyrillic_ze
C	6DB	XK_Cyrillic_sha
C	6DC	XK_Cyrillic_e
C	6DD	XK_Cyrillic_shcha
C	6DE	XK_Cyrillic_che
C	6DF	XK_Cyrillic_hardsign
C	6E0	XK_Cyrillic_YU
C	6E1	XK_Cyrillic_A
C	6E2	XK_Cyrillic_BE
C	6E3	XK_Cyrillic_TSE
C	6E4	XK_Cyrillic_DE
C	6E5	XK_Cyrillic_IE

Code	Value	Name
C	6E6	XK_Cyrillic_EF
C	6E7	XK_Cyrillic_GHE
C	6E8	XK_Cyrillic_HA
C	6E9	XK_Cyrillic_I
C	6EA	XK_Cyrillic_SHORTI
C	6EB	XK_Cyrillic_KA
C	6EC	XK_Cyrillic_EL
C	6ED	XK_Cyrillic_EM
C	6EE	XK_Cyrillic_EN
C	6EF	XK_Cyrillic_O
C	6F0	XK_Cyrillic_PE
C	6F1	XK_Cyrillic_YA
C	6F2	XK_Cyrillic_ER
C	6F3	XK_Cyrillic_ES
C	6F4	XK_Cyrillic_TE
C	6F5	XK_Cyrillic_U
C	6F6	XK_Cyrillic_ZHE
C	6F7	XK_Cyrillic_VE
C	6F8	XK_Cyrillic_SOFTSIGN
C	6F9	XK_Cyrillic_YERU
C	6FA	XK_Cyrillic_ZE
C	6FB	XK_Cyrillic_SHA
C	6FC	XK_Cyrillic_E
C	6FD	XK_Cyrillic_SHCHA
C	6FE	XK_Cyrillic_CHE
C	6FF	XK_Cyrillic_HARDSIGN
G	7A1	XK_Greek_ALPHAaccent
G	7A2	XK_Greek_EPSILONaccent
G	7A3	XK_Greek_ETAaccent
G	7A4	XK_Greek_IOTAaccent
G	7A5	XK_Greek_IOTAdiaeresis
G	7A7	XK_Greek_OMICRONaccent
G	7A8	XK_Greek_UPSILONaccent
G	7A9	XK_Greek_UPSILONdieresis
G	7AB	XK_Greek_OMEGAaccent
G	7AE	XK_Greek_accentdieresis
G	7AF	XK_Greek_horizbar
G	7B1	XK_Greek_alphaaccent
G	7B2	XK_Greek_epsilonaccent
G	7B3	XK_Greek_etaaccent
G	7B4	XK_Greek_iotaaccent
G	7B5	XK_Greek_iotadieresis
G	7B6	XK_Greek_iotaaccentdieresis
G	7B7	XK_Greek_omicronaccent
G	7B8	XK_Greek_upsilonaccent
G	7B9	XK_Greek_upsilondieresis
G	7BA	XK_Greek_upsilonaccentdieresis
G	7BB	XK_Greek_omegaaccent
G	7C1	XK_Greek_ALPHA
G	7C2	XK_Greek_BETA
G	7C3	XK_Greek_GAMMA
G	7C4	XK_Greek_DELTA
G	7C5	XK_Greek_EPSILON

Code	Value	Name
G	7C6	XK_Greek_ZETA
G	7C7	XK_Greek_ETA
G	7C8	XK_Greek_THETA
G	7C9	XK_Greek_IOTA
G	7CA	XK_Greek_KAPPA
G	7CB	XK_Greek_LAMBDA
G	7CB	XK_Greek_LAMDA
G	7CC	XK_Greek_MU
G	7CD	XK_Greek_NU
G	7CE	XK_Greek_XI
G	7CF	XK_Greek_OMICRON
G	7D0	XK_Greek_PI
G	7D1	XK_Greek_RHO
G	7D2	XK_Greek_SIGMA
G	7D4	XK_Greek_TAU
G	7D5	XK_Greek_UPSILON
G	7D6	XK_Greek_PHI
G	7D7	XK_Greek_CHI
G	7D8	XK_Greek_PSI
G	7D9	XK_Greek_OMEGA
G	7E1	XK_Greek_alpha
G	7E2	XK_Greek_beta
G	7E3	XK_Greek_gamma
G	7E4	XK_Greek_delta
G	7E5	XK_Greek_epsilon
G	7E6	XK_Greek_zeta
G	7E7	XK_Greek_eta
G	7E8	XK_Greek_theta
G	7E9	XK_Greek_iota
G	7EA	XK_Greek_kappa
G	7EB	XK_Greek_lambda
G	7EB	XK_Greek_lamda
G	7EC	XK_Greek_mu
G	7ED	XK_Greek_nu
G	7EE	XK_Greek_xi
G	7EF	XK_Greek_omicron
G	7F0	XK_Greek_pi
G	7F1	XK_Greek_rho
G	7F2	XK_Greek_sigma
G	7F3	XK_Greek_finalsmallsigma
G	7F4	XK_Greek_tau
G	7F5	XK_Greek_upsilon
G	7F6	XK_Greek_phi
G	7F7	XK_Greek_chi
G	7F8	XK_Greek_psi
G	7F9	XK_Greek_omega
T	8A1	XK_leftradical
T	8A2	XK_topleftradical
T	8A3	XK_horizconnector
T	8A4	XK_topintegral
T	8A5	XK_botintegral
T	8A6	XK_vertconnector
T	8A7	XK_topleftsqbracket

Code	Value	Name
T	8A8	XK_botleftsqbracket
T	8A9	XK_toprightsqbracket
T	8AA	XK_botrightsqbracket
T	8AB	XK_topleftparens
T	8AC	XK_botleftparens
T	8AD	XK_toprightparens
T	8AE	XK_botrightparens
T	8AF	XK_leftmiddlecurlybrace
T	8B0	XK_rightmiddlecurlybrace
T	8B1	XK_topleftsummation
T	8B2	XK_botleftsummation
T	8B3	XK_topvertsummationconnector
T	8B4	XK_botvertsummationconnector
T	8B5	XK_toprightsummation
T	8B6	XK_botrightsummation
T	8B7	XK_rightmiddlesummation
T	8BC	XK_lessthanequal
T	8BD	XK_notequal
T	8BE	XK_greaterthanequal
T	8BF	XK_integral
T	8C0	XK_therefore
T	8C1	XK_variation
T	8C2	XK_infinity
T	8C5	XK_nabla
T	8C8	XK_approximate
T	8C9	XK_similarequal
T	8CD	XK_ifonlyif
T	8CE	XK_implies
T	8CF	XK_identical
T	8D6	XK_radical
T	8DA	XK_includedin
T	8DB	XK_includes
T	8DC	XK_intersection
T	8DD	XK_union
T	8DE	XK_logicaland
T	8DF	XK_logicalor
T	8EF	XK_partialderivative
T	8F6	XK_function
T	8FB	XK_leftarrow
T	8FC	XK_uparrow
T	8FD	XK_rightarrow
T	8FE	XK_downarrow
S	9DF	XK_blank
S	9E0	XK_soliddiamond
S	9E1	XK_checkerboard
S	9E2	XK_ht
S	9E3	XK_ff
S	9E4	XK_cr
S	9E5	XK_lf
S	9E8	XK_nl
S	9E9	XK_vt
S	9EA	XK_lowrightcorner
S	9EB	XK_uprightcorner

Code	Value	Name
S	9EC	XK_upleftcorner
S	9ED	XK_lowleftcorner
S	9EE	XK_crossinglines
S	9EF	XK_horizlinescan1
S	9F0	XK_horizlinescan3
S	9F1	XK_horizlinescan5
S	9F2	XK_horizlinescan7
S	9F3	XK_horizlinescan9
S	9F4	XK_leftt
S	9F5	XK_rightt
S	9F6	XK_bott
S	9F7	XK_topt
S	9F8	XK_vertbar
P	AA1	XK_emspace
P	AA2	XK_enspace
P	AA3	XK_em3space
P	AA4	XK_em4space
P	AA5	XK_digitspace
P	AA6	XK_punctspace
P	AA7	XK_thinspace
P	AA8	XK_hairspace
P	AA9	XK_emdash
P	AAA	XK_endash
P	AAC	XK_signifblank
P	AAE	XK_ellipsis
P	AAF	XK_doubbaselinedot
P	AB0	XK_onethird
P	AB1	XK_twothirds
P	AB2	XK_onefifth
P	AB3	XK_twofifths
P	AB4	XK_threefifths
P	AB5	XK_fourfifths
P	AB6	XK_onesixth
P	AB7	XK_fivesixths
P	AB8	XK_careof
P	ABB	XK_figdash
P	ABC	XK_leftanglebracket
P	ABD	XK_decimalpoint
P	ABE	XK_rightanglebracket
P	ABF	XK_marker
P	AC3	XK_oneeighth
P	AC4	XK_threeeighths
P	AC5	XK_fiveeighths
P	AC6	XK_seveneighths
P	AC9	XK_trademark
P	ACA	XK_signaturemark
P	ACB	XK_trademarkincircle
P	ACC	XK_leftopentriangle
P	ACD	XK_rightopentriangle
P	ACE	XK_emopencircle
P	ACF	XK_emopenrectangle
P	AD0	XK_leftsinglequotemark
P	AD1	XK_rightsinglequotemark

Code	Value	Name
P	AD2	XK_leftdoublequotemark
P	AD3	XK_rightdoublequotemark
P	AD4	XK_prescription
P	AD6	XK_minutes
P	AD7	XK_seconds
P	AD9	XK_latincross
P	ADA	XK_hexagram
P	ADB	XK_filledrectbullet
P	ADC	XK_filledlefttribullet
P	ADD	XK_filledrighttribullet
P	ADE	XK_emfilledcircle
P	ADF	XK_emfilledrect
P	AE0	XK_enopencircbullet
P	AE1	XK_enopensquarebullet
P	AE2	XK_openrectbullet
P	AE3	XK_opentribulletup
P	AE4	XK_opentribulletdown
P	AE5	XK_openstar
P	AE6	XK_enfilledcircbullet
P	AE7	XK_enfilledsqbullet
P	AE8	XK_filledtribulletup
P	AE9	XK_filledtribulletdown
P	AEA	XK_leftpointer
P	AEB	XK_rightpointer
P	AEC	XK_club
P	AED	XK_diamond
P	AEE	XK_heart
P	AF0	XK_maltesecross
P	AF1	XK_dagger
P	AF2	XK_doubledagger
P	AF3	XK_checkmark
P	AF4	XK_ballotcross
P	AF5	XK_musicalsharp
P	AF6	XK_musicalflat
P	AF7	XK_malesymbol
P	AF8	XK_femalesymbol
P	AF9	XK_telephone
P	AFA	XK_telephonerecorder
P	AFB	XK_phonographcopyright
P	AFC	XK_caret
P	AFD	XK_singlelowquotemark
P	AFE	XK_doublelowquotemark
P	AFF	XK_cursor
L	BA3	XK_leftcaret
L	BA6	XK_rightcaret
L	BA8	XK_downcaret
L	BA9	XK_upcaret
L	BC0	XK_overbar
L	BC2	XK_downtack
L	BC3	XK_upshoe
L	BC4	XK_downstile
L	BC6	XK_underbar
L	BCA	XK_jot

Code	Value	Name
L	BCC	XK_quad
L	BCE	XK_uptack
L	BCF	XK_circle
L	BD3	XK_upstile
L	BD6	XK_downshoe
L	BD8	XK_rightshoe
L	BDA	XK_leftshoe
L	BDC	XK_lefttack
L	BFC	XK_righttack
H	CDF	XK_hebrew_doublelowline
H	CE0	XK_hebrew_aleph
H	CE1	XK_hebrew_bet
H	CE1	XK_hebrew_beth †
H	CE2	XK_hebrew_gimel
H	CE2	XK_hebrew_gimmel †
H	CE3	XK_hebrew_dalet
H	CE3	XK_hebrew_daleth †
H	CE4	XK_hebrew_he
H	CE5	XK_hebrew_waw
H	CE6	XK_hebrew_zain
H	CE6	XK_hebrew_zayin †
H	CE7	XK_hebrew_chet
H	CE7	XK_hebrew_het †
H	CE8	XK_hebrew_tet
H	CE8	XK_hebrew_teth †
H	CE9	XK_hebrew_yod
H	CEA	XK_hebrew_finalkaph
H	CEB	XK_hebrew_kaph
H	CEC	XK_hebrew_lamed
H	CED	XK_hebrew_finalmem
H	CEE	XK_hebrew_mem
H	CEF	XK_hebrew_finalnun
H	CF0	XK_hebrew_nun
H	CF1	XK_hebrew_samech
H	CF1	XK_hebrew_samekh †
H	CF2	XK_hebrew_ayin
H	CF3	XK_hebrew_finalpe
H	CF4	XK_hebrew_pe
H	CF5	XK_hebrew_finalzade
H	CF5	XK_hebrew_finalzadi †
H	CF6	XK_hebrew_zade
H	CF6	XK_hebrew_zadi †
H	CF7	XK_hebrew_qoph
H	CF7	XK_hebrew_kuf †
H	CF8	XK_hebrew_resh
H	CF9	XK_hebrew_shin
H	CFA	XK_hebrew_taw
H	CFA	XK_hebrew_taf †
M	FF08	XK_BackSpace
M	FF09	XK_Tab
M	FF0A	XK_Linefeed
M	FF0B	XK_Clear
M	FF0D	XK_Return

Code	Value	Name
M	FF13	XK_Pause
M	FF14	XK_Scroll_Lock
M	FF1B	XK_Escape
M	FF20	XK_Multi_key
M	FF21	XK_Kanji
M	FF22	XK_Muhenkan
M	FF23	XK_Henkan_Mode
M	FF23	XK_Henkan
M	FF24	XK_Romaji
M	FF25	XK_Hiragana
M	FF26	XK_Katakana
M	FF27	XK_Hiragana_Katakana
M	FF28	XK_Zenkaku
M	FF29	XK_Hankaku
M	FF2A	XK_Zenkaku_Hankaku
M	FF2B	XK_Touroku
M	FF2C	XK_Massyo
M	FF2D	XK_Kana_Lock
M	FF2E	XK_Kana_Shift
M	FF2F	XK_Eisu_Shift
M	FF30	XK_Eisu_toggle
M	FF50	XK_Home
M	FF51	XK_Left
M	FF52	XK_Up
M	FF53	XK_Right
M	FF54	XK_Down
M	FF55	XK_Prior
M	FF56	XK_Next
M	FF57	XK_End
M	FF58	XK_Begin
M	FF60	XK_Select
M	FF61	XK_Print
M	FF62	XK_Execute
M	FF63	XK_Insert
M	FF65	XK_Undo
M	FF66	XK_Redo
M	FF67	XK_Menu
M	FF68	XK_Find
M	FF69	XK_Cancel
M	FF6A	XK_Help
M	FF6B	XK_Break
A	FF7E	XK_Arabic_switch
G	FF7E	XK_Greek_switch
H	FF7E	XK_Hebrew_switch
M	FF7E	XK_Mode_switch
K	FF7E	XK_kana_switch
M	FF7E	XK_script_switch
M	FF7F	XK_Num_Lock
M	FF80	XK_KP_Space
M	FF89	XK_KP_Tab
M	FF8D	XK_KP_Enter
M	FF91	XK_KP_F1
M	FF92	XK_KP_F2

Code	Value	Name
M	FF93	XK_KP_F3
M	FF94	XK_KP_F4
M	FFAA	XK_KP_Multiply
M	FFAB	XK_KP_Add
M	FFAC	XK_KP_Separator
M	FFAD	XK_KP_Subtract
M	FFAE	XK_KP_Decimal
M	FFAF	XK_KP_Divide
M	FFB0	XK_KP_0
M	FFB1	XK_KP_1
M	FFB2	XK_KP_2
M	FFB3	XK_KP_3
M	FFB4	XK_KP_4
M	FFB5	XK_KP_5
M	FFB6	XK_KP_6
M	FFB7	XK_KP_7
M	FFB8	XK_KP_8
M	FFB9	XK_KP_9
M	FFBD	XK_KP_Equal
M	FFBE	XK_F1
M	FFBF	XK_F2
M	FFC0	XK_F3
M	FFC1	XK_F4
M	FFC2	XK_F5
M	FFC3	XK_F6
M	FFC4	XK_F7
M	FFC5	XK_F8
M	FFC6	XK_F9
M	FFC7	XK_F10
M	FFC8	XK_F11
M	FFC8	XK_L1
M	FFC9	XK_F12
M	FFC9	XK_L2
M	FFCA	XK_F13
M	FFCA	XK_L3
M	FFCB	XK_F14
M	FFCB	XK_L4
M	FFCC	XK_F15
M	FFCC	XK_L5
M	FFCD	XK_F16
M	FFCD	XK_L6
M	FFCE	XK_F17
M	FFCE	XK_L7
M	FFCF	XK_F18
M	FFCF	XK_L8
M	FFD0	XK_F19
M	FFD0	XK_L9
M	FFD1	XK_F20
M	FFD1	XK_L10
M	FFD2	XK_F21
M	FFD2	XK_R1
M	FFD3	XK_F22
M	FFD3	XK_R2

Code	Value	Name
M	FFD4	XK_F23
M	FFD4	XK_R3
M	FFD5	XK_F24
M	FFD5	XK_R4
M	FFD6	XK_F25
M	FFD6	XK_R5
M	FFD7	XK_F26
M	FFD7	XK_R6
M	FFD8	XK_F27
M	FFD8	XK_R7
M	FFD9	XK_F28
M	FFD9	XK_R8
M	FFDA	XK_F29
M	FFDA	XK_R9
M	FFDB	XK_F30
M	FFDB	XK_R10
M	FFDC	XK_F31
M	FFDC	XK_R11
M	FFDD	XK_F32
M	FFDD	XK_R12
M	FFDE	XK_F33
M	FFDE	XK_R13
M	FFDF	XK_F34
M	FFDF	XK_R14
M	FFE0	XK_F35
M	FFE0	XK_R15
M	FFE1	XK_Shift_L
M	FFE2	XK_Shift_R
M	FFE3	XK_Control_L
M	FFE4	XK_Control_R
M	FFE5	XK_Caps_Lock
M	FFE6	XK_Shift_Lock
M	FFE7	XK_Meta_L
M	FFE8	XK_Meta_R
M	FFE9	XK_Alt_L
M	FFEA	XK_Alt_R
M	FFEB	XK_Super_L
M	FFEC	XK_Super_R
M	FFED	XK_Hyper_L
M	FFEE	XK_Hyper_R
M	FFFF	XK_Delete
-	FFFFFF	XK_VoidSymbol

18.3   Color Names

The functions *XLookupColor*( ) and *XAllocNamedColor*( ) convert a color name to an RGB triple. X/Open-compliant systems are required to recognize certain names and generate RGB triples from them, but this document does not specify the RGB triples generated.

The names that must be recognized are specified in the following table. The following rules apply to all X/Open-compliant systems:

- If a color name is recognized, then all other names on the same line of the table must also be recognized, and must generate the same RGB triple.

- If two color names on different lines of the table are both recognized, then they must generate different ''screen'' RGB triples (that is, if both triples are displayed at the same time, they must be visually distinct).

- All names in the table are shown in ISO Latin-1 lower-case, with spaces between words. Each name shall also be recognized with the first letter of each word being upper-case, and no spaces between words. For example, ''light red'' shall also be recognized when spelt ''LightRed''.

- Those names marked with an A must be recognized.

- Those names marked with an M must be recognized if the color map specified belongs to a *GrayScale* or *StaticGray* Visual, and has at least 4 color cells.

- Those names marked with a C must be recognized if the color map specified belongs to a *PseudoColor*, *StaticColor*, *TrueColor* or *DirectColor* Visual.

- A system that recognizes any name marked with an M, a V or an X for a given color map must recognize all names marked with a C, an M or a V for that color map; it need not recognize all names marked with an X.

Set	Color name(s)	
A	black	
A	white	
C	blue	
C	cyan	
C	green	
C	magenta	
C	red	
C	yellow	
M	gray	grey
M	dark gray	dark grey
V	dark blue	
V	brown	
V	dark cyan	
V	dark green	
V	dark magenta	
V	dark red	
X	medium blue	
X	midnight blue	

Set	Color name(s)	
X	navy blue	
X	sky blue	
X	coral	
X	gold	
X	dark slate gray	dark slate grey
X	dim gray	dim grey
X	light gray	light grey
X	dark green	
X	forest green	
X	lime green	
X	pale green	
X	spring green	
X	maroon	
X	orange	
X	pink	
X	indian red	
X	orange red	
X	violet red	
X	salmon	
X	sienna	
X	tan	
X	turquoise	
X	violet	
X	blue violet	
X	wheat	

Xlib Functions and Protocol Requests

This appendix provides two tables that relate to Xlib functions and the X protocol. The following table lists each Xlib function (in alphabetical order) and the corresponding protocol request that it generates.

Xlib Function	Protocol Request
XActivateScreenSaver	ForceScreenSaver
XAddHost	ChangeHosts
XAddHosts	ChangeHosts
XAddToSaveSet	ChangeSaveSet
XAllocColor	AllocColor
XAllocColorCells	AllocColorCells
XAllocColorPlanes	AllocColorPlanes
XAllocNamedColor	AllocNamedColor
XAllowEvents	AllowEvents
XAutoRepeatOff	ChangeKeyboardControl
XAutoRepeatOn	ChangeKeyboardControl
XBell	Bell
XChangeActivePointerGrab	ChangeActivePointerGrab
XChangeGC	ChangeGC
XChangeKeyboardControl	ChangeKeyboardControl
XChangeKeyboardMapping	ChangeKeyboardMapping
XChangePointerControl	ChangePointerControl
XChangeProperty	ChangeProperty
XChangeSaveSet	ChangeSaveSet
XChangeWindowAttributes	ChangeWindowAttributes
XCirculateSubwindows	CirculateWindow
XCirculateSubwindowsDown	CirculateWindow
XCirculateSubwindowsUp	CirculateWindow
XClearArea	ClearArea
XClearWindow	ClearArea
XConfigureWindow	ConfigureWindow
XConvertSelection	ConvertSelection
XCopyArea	CopyArea
XCopyColormapAndFree	CopyColormapAndFree
XCopyGC	CopyGC
XCopyPlane	CopyPlane
XCreateBitmapFromData	CreateGC
	CreatePixmap
	FreeGC
	PutImage
XCreateColormap	CreateColormap
XCreateFontCursor	CreateGlyphCursor
XCreateGC	CreateGC

Xlib Function	Protocol Request
XCreateGlyphCursor	CreateGlyphCursor
XCreatePixmap	CreatePixmap
XCreatePixmapCursor	CreateCursor
XCreatePixmapFromData	CreateGC
	CreatePixmap
	FreeGC
	PutImage
XCreateSimpleWindow	CreateWindow
XCreateWindow	CreateWindow
XDefineCursor	ChangeWindowAttributes
XDeleteProperty	DeleteProperty
XDestroySubwindows	DestroySubwindows
XDestroyWindow	DestroyWindow
XDisableAccessControl	SetAccessControl
XDrawArc	PolyArc
XDrawArcs	PolyArc
XDrawImageString	ImageText8
XDrawImageString16	ImageText16
XDrawLine	PolySegment
XDrawLines	PolyLine
XDrawPoint	PolyPoint
XDrawPoints	PolyPoint
XDrawRectangle	PolyRectangle
XDrawRectangles	PolyRectangle
XDrawSegments	PolySegment
XDrawString	PolyText8
XDrawString16	PolyText16
XDrawText	PolyText8
XDrawText16	PolyText16
XEnableAccessControl	SetAccessControl
XFetchBytes	GetProperty
XFetchName	GetProperty
XFillArc	PolyFillArc
XFillArcs	PolyFillArc
XFillPolygon	FillPoly
XFillRectangle	PolyFillRectangle
XFillRectangles	PolyFillRectangle
XForceScreenSaver	ForceScreenSaver
XFreeColormap	FreeColormap
XFreeColors	FreeColors
XFreeCursor	FreeCursor
XFreeFont	CloseFont
XFreeGC	FreeGC
XFreePixmap	FreePixmap
XGetAtomName	GetAtomName
XGetClassHint	GetProperty
XGetFontPath	GetFontPath

Xlib Function	Protocol Request
XGetGeometry	GetGeometry
XGetIconName	GetProperty
XGetIconSizes	GetProperty
XGetImage	GetImage
XGetInputFocus	GetInputFocus
XGetKeyboardControl	GetKeyboardControl
XGetKeyboardMapping	GetKeyboardMapping
XGetModifierMapping	GetModifierMapping
XGetMotionEvents	GetMotionEvents
XGetModifierMapping	GetModifierMapping
XGetNormalHints	GetProperty
XGetPointerControl	GetPointerControl
XGetPointerMapping	GetPointerMapping
XGetRGBColormaps	GetProperty
XGetScreenSaver	GetScreenSaver
XGetSelectionOwner	GetSelectionOwner
XGetSizeHints	GetProperty
XGetTextProperty	GetProperty
XGetTransientForHint	GetProperty
XGetWMClientMachine	GetProperty
XGetWMColormapWindows	GetProperty
	InternAtom
XGetWMHints	GetProperty
XGetWMIconName	GetProperty
XGetWMName	GetProperty
XGetWMNormalHints	GetProperty
XGetWMProtocols	GetProperty
	InternAtom
XGetWMSizeHints	GetProperty
XGetWindowAttributes	GetWindowAttributes
	GetGeometry
XGetWindowProperty	GetProperty
XGetZoomHints	GetProperty
XGrabButton	GrabButton
XGrabKey	GrabKey
XGrabKeyboard	GrabKeyboard
XGrabPointer	GrabPointer
XGrabServer	GrabServer
XIconifyWindow	InternAtom
	SendEvent
XInitExtension	QueryExtension
XInstallColormap	InstallColormap
XInternAtom	InternAtom
XKillClient	KillClient
XListExtensions	ListExtensions
XListFonts	ListFonts
XListFontsWithInfo	ListFontsWithInfo

Xlib Function	Protocol Request
XListHosts	ListHosts
XListInstalledColormaps	ListInstalledColormaps
XListProperties	ListProperties
XLoadFont	OpenFont
XLoadQueryFont	OpenFont
	QueryFont
XLookupColor	LookupColor
XLowerWindow	ConfigureWindow
XMapRaised	ConfigureWindow
	MapWindow
XMapSubwindows	MapSubwindows
XMapWindow	MapWindow
XMoveResizeWindow	ConfigureWindow
XMoveWindow	ConfigureWindow
XNoOp	NoOperation
XOpenDisplay	CreateGC
XParseColor	LookupColor
XPutImage	PutImage
XQueryBestCursor	QueryBestSize
XQueryBestSize	QueryBestSize
XQueryBestStipple	QueryBestSize
XQueryBestTile	QueryBestSize
XQueryColor	QueryColors
XQueryColors	QueryColors
XQueryExtension	QueryExtension
XQueryFont	QueryFont
XQueryKeymap	QueryKeymap
XQueryPointer	QueryPointer
XQueryTextExtents	QueryTextExtents
XQueryTextExtents16	QueryTextExtents
XQueryTree	QueryTree
XRaiseWindow	ConfigureWindow
XReadBitmapFile	CreateGC
	CreatePixmap
	FreeGC
	PutImage
XRecolorCursor	RecolorCursor
XReconfigureWMWindow	ConfigureWindow
	SendEvent
XRemoveFromSaveSet	ChangeSaveSet
XRemoveHost	ChangeHosts
XRemoveHosts	ChangeHosts
XReparentWindow	ReparentWindow
XResetScreenSaver	ForceScreenSaver
XResizeWindow	ConfigureWindow
XRestackWindows	ConfigureWindow
XRotateBuffers	RotateProperties

Xlib Function	Protocol Request
XRotateWindowProperties	RotateProperties
XSelectInput	ChangeWindowAttributes
XSendEvent	SendEvent
XSetAccessControl	SetAccessControl
XSetArcMode	ChangeGC
XSetBackground	ChangeGC
XSetClassHint	ChangeProperty
XSetClipMask	ChangeGC
XSetClipOrigin	ChangeGC
XSetClipRectangles	SetClipRectangles
XSetCloseDownMode	SetCloseDownMode
XSetCommand	ChangeProperty
XSetDashes	SetDashes
XSetFillRule	ChangeGC
XSetFillStyle	ChangeGC
XSetFont	ChangeGC
XSetFontPath	SetFontPath
XSetForeground	ChangeGC
XSetFunction	ChangeGC
XSetGraphicsExposures	ChangeGC
XSetIconName	ChangeProperty
XSetIconSizes	ChangeProperty
XSetInputFocus	SetInputFocus
XSetLineAttributes	ChangeGC
XSetModifierMapping	SetModifierMapping
XSetNormalHints	ChangeProperty
XSetPlaneMask	ChangeGC
XSetPointerMapping	SetPointerMapping
XSetRGBColormaps	ChangeProperty
XSetScreenSaver	SetScreenSaver
XSetSelectionOwner	SetSelectionOwner
XSetSizeHints	ChangeProperty
XSetStandardProperties	ChangeProperty
XSetState	ChangeGC
XSetStipple	ChangeGC
XSetSubwindowMode	ChangeGC
XSetTextProperty	ChangeProperty
XSetTile	ChangeGC
XSetTransientForHint	ChangeProperty
XSetTSOrigin	ChangeGC
XSetWMClientMachine	ChangeProperty
XSetWMColormapWindows	ChangeProperty
	InternAtom
XSetWMHints	ChangeProperty
XSetWMIconName	ChangeProperty
XSetWMName	ChangeProperty
XSetWMNormalHints	ChangeProperty

Xlib Function	Protocol Request
XSetWMProperties	ChangeProperty
XSetWMProtocols	ChangeProperty
	InternAtom
XSetWMSizeHints	ChangeProperty
XSetWindowBackground	ChangeWindowAttributes
XSetWindowBackgroundPixmap	ChangeWindowAttributes
XSetWindowBorder	ChangeWindowAttributes
XSetWindowBorderPixmap	ChangeWindowAttributes
XSetWindowBorderWidth	ConfigureWindow
XSetWindowColormap	ChangeWindowAttributes
XSetZoomHints	ChangeProperty
XStoreBuffer	ChangeProperty
XStoreBytes	ChangeProperty
XStoreColor	StoreColors
XStoreColors	StoreColors
XStoreName	ChangeProperty
XStoreNamedColor	StoreNamedColor
XSync	GetInputFocus
XSynchronize	GetInputFocus
XTranslateCoordinates	TranslateCoordinates
XUndefineCursor	ChangeWindowAttributes
XUngrabButton	UngrabButton
XUngrabKey	UngrabKey
XUngrabKeyboard	UngrabKeyboard
XUngrabPointer	UngrabPointer
XUngrabServer	UngrabServer
XUninstallColormap	UninstallColormap
XUnloadFont	CloseFont
XUnmapSubwindows	UnmapSubwindows
XUnmapWindow	UnmapWindow
XWarpPointer	WarpPointer
XWithdrawWindow	SendEvent
	UnmapWindow

The following table lists each X protocol request (in alphabetical order) and the Xlib functions that reference it.

Protocol Request	Xlib Function
AllocColor	XAllocColor
AllocColorCells	XAllocColorCells
AllocColorPlanes	XAllocColorPlanes
AllocNamedColor	XAllocNamedColor
AllowEvents	XAllowEvents
Bell	XBell
SetAccessControl	XDisableAccessControl
	XEnableAccessControl
	XSetAccessControl

Protocol Request	Xlib Function
ChangeActivePointerGrab	XChangeActivePointerGrab
SetCloseDownMode	XSetCloseDownMode
ChangeGC	XChangeGC
	XSetArcMode
	XSetBackground
	XSetClipMask
	XSetClipOrigin
	XSetFillRule
	XSetFillStyle
	XSetFont
	XSetForeground
	XSetFunction
	XSetGraphicsExposures
	XSetLineAttributes
	XSetPlaneMask
	XSetState
	XSetStipple
	XSetSubwindowMode
	XSetTile
	XSetTSOrigin
ChangeHosts	XAddHost
	XAddHosts
	XRemoveHost
	XRemoveHosts
ChangeKeyboardControl	XAutoRepeatOff
	XAutoRepeatOn
	XChangeKeyboardControl
ChangeKeyboardMapping	XChangeKeyboardMapping
ChangePointerControl	XChangePointerControl
ChangeProperty	XChangeProperty
	XSetClassHint
	XSetCommand
	XSetIconName
	XSetIconSizes
	XSetNormalHints
	XSetRGBColormaps
	XSetSizeHints
	XSetStandardProperties
	XSetTextProperty
	XSetTransientForHint
	XSetWMClientMachine
	XSetWMColormapWindows
	XSetWMHints
	XSetWMIconName
	XSetWMName
	XSetWMNormalHints
	XSetWMProperties

Protocol Request	Xlib Function
	XSetWMProtocols
	XSetWMSizeHints
	XSetZoomHints
	XStoreBuffer
	XStoreBytes
	XStoreName
ChangeSaveSet	XAddToSaveSet
	XChangeSaveSet
	XRemoveFromSaveSet
ChangeWindowAttributes	XChangeWindowAttributes
	XDefineCursor
	XSelectInput
	XSetWindowBackground
	XSetWindowBackgroundPixmap
	XSetWindowBorder
	XSetWindowBorderPixmap
	XSetWindowColormap
	XUndefineCursor
CirculateWindow	XCirculateSubwindowsDown
	XCirculateSubwindowsUp
	XCirculateSubwindows
ClearArea	XClearArea
	XClearWindow
CloseFont	XFreeFont
	XUnloadFont
ConfigureWindow	XConfigureWindow
	XLowerWindow
	XMapRaised
	XMoveResizeWindow
	XMoveWindow
	XRaiseWindow
	XReconfigureWMWindow
	XResizeWindow
	XRestackWindows
	XSetWindowBorderWidth
ConvertSelection	XConvertSelection
CopyArea	XCopyArea
CopyColormapAndFree	XCopyColormapAndFree
CopyGC	XCopyGC
CopyPlane	XCopyPlane
CreateColormap	XCreateColormap
CreateCursor	XCreatePixmapCursor
CreateGC	XCreateGC
	XCreateBitmapFromData
	XCreatePixmapFromData
	XOpenDisplay
	XReadBitmapFile

Protocol Request	Xlib Function
CreateGlyphCursor	XCreateFontCursor
	XCreateGlyphCursor
CreatePixmap	XCreatePixmap
	XCreateBitmapFromData
	XCreatePixmapFromData
	XReadBitmapFile
CreateWindow	XCreateSimpleWindow
	XCreateWindow
DeleteProperty	XDeleteProperty
DestroySubwindows	XDestroySubwindows
DestroyWindow	XDestroyWindow
FillPoly	XFillPolygon
ForceScreenSaver	XActivateScreenSaver
	XForceScreenSaver
	XResetScreenSaver
FreeColormap	XFreeColormap
FreeColors	XFreeColors
FreeCursor	XFreeCursor
FreeGC	XFreeGC
	XCreateBitmapFromData
	XCreatePixmapFromData
	XReadBitmapFile
FreePixmap	XFreePixmap
GetAtomName	XGetAtomName
GetFontPath	XGetFontPath
GetGeometry	XGetGeometry
	XGetWindowAttributes
GetImage	XGetImage
GetInputFocus	XGetInputFocus
	XSync
	XSynchronize
GetKeyboardControl	XGetKeyboardControl
GetKeyboardMapping	XGetKeyboardMapping
GetModifierMapping	XGetModifierMapping
GetMotionEvents	XGetMotionEvents
GetPointerControl	XGetPointerControl
GetPointerMapping	XGetPointerMapping
GetProperty	XFetchBytes
	XFetchName
	XGetClassHint
	XGetIconName
	XGetIconSizes
	XGetNormalHints
	XGetRGBColormaps
	XGetSizeHints
	XGetTextProperty
	XGetTransientForHint

Stamp:XXXXXXXXXXXXXXXXXXXXXXXXX

Protocol Request	Xlib Function
	XGetWMClientMachine
	XGetWMColormapWindows
	XGetWMHints
	XGetWMIconName
	XGetWMName
	XGetWMNormalHints
	XGetWMProtocols
	XGetWMSizeHints
	XGetWindowProperty
	XGetZoomHints
GetSelectionOwner	XGetSelectionOwner
GetWindowAttributes	XGetWindowAttributes
GrabButton	XGrabButton
GrabKey	XGrabKey
GrabKeyboard	XGrabKeyboard
GrabPointer	XGrabPointer
GrabServer	XGrabServer
ImageText16	XDrawImageString16
ImageText8	XDrawImageString
InstallColormap	XInstallColormap
InternAtom	XGetWMColormapWindows
	XGetWMProtocols
	XIconifyWindow
	XInternAtom
	XSetWMColormapWindows
	XSetWMProtocols
KillClient	XKillClient
ListExtensions	XListExtensions
ListFonts	XListFonts
ListFontsWithInfo	XListFontsWithInfo
ListHosts	XListHosts
ListInstalledColormaps	XListInstalledColormaps
ListProperties	XListProperties
LookupColor	XLookupColor
	XParseColor
MapSubwindows	XMapSubwindows
MapWindow	XMapRaised
	XMapWindow
NoOperation	XNoOp
OpenFont	XLoadFont
	XLoadQueryFont
PolyArc	XDrawArc
	XDrawArcs
PolyFillArc	XFillArc
	XFillArcs
PolyFillRectangle	XFillRectangle
	XFillRectangles

Protocol Request	Xlib Function
PolyLine	XDrawLines
PolyPoint	XDrawPoint
	XDrawPoints
PolyRectangle	XDrawRectangle
	XDrawRectangles
PolySegment	XDrawLine
	XDrawSegments
PolyText16	XDrawString16
	XDrawText16
PolyText8	XDrawString
	XDrawText
PutImage	XPutImage
	XCreateBitmapFromData
	XCreatePixmapFromData
	XReadBitmapFile
QueryBestSize	XQueryBestCursor
	XQueryBestSize
	XQueryBestStipple
	XQueryBestTile
QueryColors	XQueryColor
	XQueryColors
QueryExtension	XInitExtension
	XQueryExtension
QueryFont	XLoadQueryFont
	XQueryFont
QueryKeymap	XQueryKeymap
QueryPointer	XQueryPointer
QueryTextExtents	XQueryTextExtents
	XQueryTextExtents16
QueryTree	XQueryTree
RecolorCursor	XRecolorCursor
ReparentWindow	XReparentWindow
RotateProperties	XRotateBuffers
	XRotateWindowProperties
SendEvent	XIconifyWindow
	XReconfigureWMWindow
	XSendEvent
	XWithdrawWindow
SetClipRectangles	XSetClipRectangles
SetCloseDownMode	XSetCloseDownMode
SetDashes	XSetDashes
SetFontPath	XSetFontPath
SetInputFocus	XSetInputFocus
SetModifierMapping	XSetModifierMapping
SetPointerMapping	XSetPointerMapping
SetScreenSaver	XGetScreenSaver
	XSetScreenSaver

Protocol Request	Xlib Function
SetSelectionOwner	XSetSelectionOwner
StoreColors	XStoreColor
	XStoreColors
StoreNamedColor	XStoreNamedColor
TranslateCoordinates	XTranslateCoordinates
UngrabButton	XUngrabButton
UngrabKey	XUngrabKey
UngrabKeyboard	XUngrabKeyboard
UngrabPointer	XUngrabPointer
UngrabServer	XUngrabServer
UninstallColormap	XUninstallColormap
UnmapSubwindows	XUnmapSubWindows
UnmapWindow	XUnmapWindow
	XWithdrawWindow
WarpPointer	XWarpPointer

The following are the available cursors that can be used with *XCreateFontCursor*.

```
#define XC_X_cursor 0                 #define XC_ll_angle 76
#define XC_arrow 2                    #define XC_lr_angle 78
#define XC_based_arrow_down 4         #define XC_man 80
#define XC_based_arrow_up 6           #define XC_middlebutton 82
#define XC_boat 8                     #define XC_mouse 84
#define XC_bogosity 10                #define XC_pencil 86
#define XC_bottom_left_corner 12      #define XC_pirate 88
#define XC_bottom_right_corner 14     #define XC_plus 90
#define XC_bottom_side 16             #define XC_question_arrow 92
#define XC_bottom_tee 18              #define XC_right_ptr 94
#define XC_box_spiral 20              #define XC_right_side 96
#define XC_center_ptr 22              #define XC_right_tee 98
#define XC_circle 24                  #define XC_rightbutton 100
#define XC_clock 26                   #define XC_rtl_logo 102
#define XC_coffee_mug 28              #define XC_sailboat 104
#define XC_cross 30                   #define XC_sb_down_arrow 106
#define XC_cross_reverse 32           #define XC_sb_h_double_arrow 108
#define XC_crosshair 34               #define XC_sb_left_arrow 110
#define XC_diamond_cross 36           #define XC_sb_right_arrow 112
#define XC_dot 38                     #define XC_sb_up_arrow 114
#define XC_dot_box_mask 40            #define XC_sb_v_double_arrow 116
#define XC_double_arrow 42            #define XC_shuttle 118
#define XC_draft_large 44             #define XC_sizing 120
#define XC_draft_small 46             #define XC_spider 122
#define XC_draped_box 48              #define XC_spraycan 124
#define XC_exchange 50                #define XC_star 126
#define XC_fleur 52                   #define XC_target 128
#define XC_gobbler 54                 #define XC_tcross 130
#define XC_gumby 56                   #define XC_top_left_arrow 132
#define XC_hand1 58                   #define XC_top_left_corner 134
#define XC_hand2 60                   #define XC_top_right_corner 136
#define XC_heart 62                   #define XC_top_side 138
#define XC_icon 64                    #define XC_top_tee 140
#define XC_iron_cross 66              #define XC_trek 142
#define XC_left_ptr 68                #define XC_ul_angle 144
#define XC_left_side 70               #define XC_umbrella 146
#define XC_left_tee 72                #define XC_ur_angle 148
#define XC_leftbutton 74              #define XC_watch 150
                                      #define XC_xterm 152
```

Extensions

Because X can evolve by extensions to the core protocol, it is important that extensions not be perceived as second class citizens. At some point, your favourite extensions may be adopted as additional parts of the X Standard.

Therefore, there should be little to distinguish the use of an extension from that of the core protocol. To avoid having to initialize extensions explicitly in application programs, it is also important that extensions perform ''lazy evaluations'' and automatically initialize themselves when called for the first time.

This appendix describes techniques for writing extensions to Xlib that will run at essentially the same performance as the core protocol requests.

Note: It is expected that a given extension to X consists of multiple requests. Defining ten new features as ten separate extensions is a bad practice. Rather, they should be packaged into a single extension and should use minor opcodes to distinguish the requests.

The symbols and macros used for writing stubs to Xlib are listed in **<X11/Xlibint.h>**.

Basic Protocol Support Routines

The basic protocol requests for extensions are *XQueryExtension* and *XListExtensions*.

```
Bool XQueryExtension(display, name, major_opcode_return,
        first_event_return, first_error_return)
    Display *display;
    char *name;
    int *major_opcode_return;
    int *first_event_return;
    int *first_error_return;
```

display   Specifies the connection to the X server.

name   Specifies the extension name.

major_opcode_return
                        Returns the major opcode.

first_event_return   Returns the first event code, if any.

Specifies the extension list.

XQueryExtension determines if the named extension is present. If the extension is not present, *False* is returned; otherwise *True* is returned. If the extension is present, the major opcode for the extension is returned to major_opcode_return; otherwise, zero is returned. Any minor opcode and the request formats are specific to the extension. If the extension involves additional event types, the base event type code is returned to first_event_return; otherwise, zero is returned. The format of the events is specific to the extension. If the extension involves additional error codes, the base error code is returned to first_error_return; otherwise, zero is returned. The format of additional data in the errors is specific to the extension.

If the extension name is not in the Host Portable Character Encoding the result is implementation-dependent. Case matters; the strings *thing*, *Thing* and *thinG* are all considered different names.

```
char **XListExtensions(display, nextensions_return)
      Display *display;
      int *nextensions_return;
```

display   Specifies the connection to the X server.

nextensions_return Returns the number of extensions listed.

XListExtensions returns a list of all extensions supported by the server. If the data returned by the server is in the Latin Portable Character Encoding, then the returned strings are in the Host Portable Character Encoding. Otherwise, the result is implementation-dependent.

```
XFreeExtensionList(list)
      char **list;
```

list   Specifies the list of extension names.

XFreeExtensionList frees the memory allocated by *XListExtensions*.

Hooking into Xlib

These functions allow you to hook into the library. They are not normally used by application programmers but are used by people who need to extend the core X protocol and the X library interface. The functions, which generate protocol requests for X, are typically called stubs.

In extensions, stubs first should check to see if they have initialized themselves on a connection. If they have not, they then should call *XInitExtension* to attempt to initialize themselves on the connection.

If the extension needs to be informed of GC/font allocation or deallocation or if the extension defines new event types, the functions described here allow the extension to be called when these events occur.

The *XExtCodes* structure returns the information from *XInitExtension* and is defined in **<X11/Xlib.h>**:

```
typedef struct _XExtCodes { /* public to extension, cannot be changed */
      int extension;          /* extension number */
      int major_opcode;       /* major op-code assigned by server */
      int first_event;        /* first event number for the extension */
      int first_error;        /* first error number for the extension */
} XExtCodes;
```

```
  XExtCodes *XInitExtension(display, name)
        Display *display;
        char *name;
```

display   Specifies the connection to the X server.

name   Specifies the extension name.

XInitExtension determines if the named extension exists. Then, it allocates storage for maintaining the information about the extension on the connection, chains this onto the extension list for the connection, and returns the information the stub implementor will need to access the extension. If the extension does not exist, *XInitExtension* returns NULL.

If the extension name is not in the Host Portable Character Encoding the result is implementation-dependent. Case matters; the strings *thing*, *Thing* and *thinG* are all considered different names.

The extension number in the *XExtCodes* structure is needed in the other calls that follow. This extension number is unique only to a single connection.

```
XExtCodes *XAddExtension(display)
        Display *display;
```

display   Specifies the connection to the X server.

For local Xlib extensions, *XAddExtension* allocates the *XExtCodes* structure, bumps the extension number count, and chains the extension onto the extension list. (This permits extensions to Xlib without requiring server extensions.)

Hooks into the Library

These functions allow you to define procedures that are to be called when various circumstances occur. The procedures include the creation of a new GC for a connection, the copying of a GC, the freeing of a GC, the creating and freeing of fonts, the conversion of events defined by extensions to and from wire format, and the handling of errors.

All of these functions return the previous routine defined for this extension.

```
int (*XESetCloseDisplay(display, extension, proc))()
      Display *display;
      int extension;
      int (*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when the display is closed.

You use this procedure to define a procedure to be called whenever *XCloseDisplay* is called. This procedure returns any previously defined procedure, usually NULL.

When *XCloseDisplay* is called, your routine is called with these arguments:

```
(*proc)(display, codes)
      Display *display;
      XExtCodes *codes;

int (*XESetCreateGC(display, extension, proc))()
      Display *display;
      int extension;
      int (*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when a GC is closed.

You use this procedure to define a procedure to be called whenever a new GC is created. This procedure returns any previously defined procedure, usually NULL.

When a GC is created, your routine is called with these arguments:

```
(*proc)(display, gc, codes)
      Display *display;
      GC gc;
      XExtCodes *codes;
```

```
int (*XESetCopyGC(display, extension, proc))()
      Display *display;
      int extension;
      int (*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when GC components are copied.

You use this procedure to define a procedure to be called whenever a GC is copied. This procedure returns any previously defined procedure, usually NULL.

When a GC is copied, your routine is called with these arguments:

```
(*proc)(display, gc, codes)
      Display *display;
      GC gc;
      XExtCodes *codes;

int (*XESetFreeGC(display, extension, proc))()
      Display *display;
      int extension;
      int (*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when a GC is freed.

You use this procedure to define a procedure to be called whenever a GC is freed. This procedure returns any previously defined procedure, usually NULL.

When a GC is freed, your routine is called with these arguments:

```
(*proc)(display, gc, codes)
      Display *display;
      GC gc;
      XExtCodes *codes;

int (*XESetCreateFont(display, extension, proc))()
      Display *display;
      int extension;
      int (*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when a font is created.

You use this procedure to define a procedure to be called whenever *XLoadQueryFont* and *XQueryFont* are called. This procedure returns any previously defined procedure, usually NULL.

When *XLoadQueryFont* or *XQueryFont* is called, your routine is called with these arguments:

```
(*proc)(display, fs, codes)
      Display *display;
      XFontStruct *fs;
      XExtCodes *codes;
int (*XESetFreeFont(display, extension, proc))()
      Display *display;
      int extension;
      int (*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when a font is freed.

You use this procedure to define a procedure to be called whenever *XFreeFont* is called. This procedure returns any previously defined procedure, usually NULL.

When *XFreeFont* is called, your routine is called with these arguments:

```
(*proc)(display, fs, codes)
      Display *display;
      XFontStruct *fs;
      XExtCodes *codes;
```

The next two functions allow you to define new events to the library. An *XEvent* structure always has a type code (type int) as the first component. This uniquely identifies what kind of event it is. The second component is always the serial number (type unsigned long) of the last request processed by the server. The third component is always a boolean (type Bool) indicating whether the event came from a *SendEvent* protocol request. The fourth component is always a pointer to the display the event was read from. The fifth component is always a resource ID of one kind or another, usually a window, carefully selected to be useful to toolkit dispatchers. The fifth component should always exist, even if the event does not have a natural ''destination''; if there is no value from the protocol to put in this component, initialize it to zero.

Note: There is an implementation limit such that your host event structure size cannot be bigger than the size of the *XEvent* union of structures. There also is no way to guarantee that more than 24 elements or 96 characters in the structure will be fully portable between machines.

```
int (*XESetWireToEvent(display, event_number, proc))()
      Display *display;
      int event_number;
      Status (*proc)();
```

display   Specifies the connection to the X server.

event_number   Specifies the event code.

proc   Specifies the routine to call when converting an event.

You use this procedure to define a procedure to be called when an event needs to be converted from wire format (*xEvent*) to host format (*XEvent*). The event number defines which protocol event number to install a conversion routine for. This procedure returns any previously defined procedure.

Note: You can replace a core event conversion routine with one of your own, although this is not encouraged. It would, however, allow you to intercept a core event and modify it before being placed in the queue or otherwise examined.

When Xlib needs to convert an event from wire format to host format, your routine is called with these arguments:

```
Status (*proc)(display, re, event)
      Display *display;
      XEvent *re;
      xEvent *event;
```

Your routine must return status to indicate if the conversion succeeded. The re argument is a pointer to where the host format event should be stored, and the event argument is the 32-byte wire event structure. In the *XEvent* structure you are creating, you must fill in the five required members of the event structure. You should fill in the type member with the type specified for the *xEvent* structure. You should copy all other members from the *xEvent* structure (wire format) to the *XEvent* structure (host format). Your conversion routine should return *True* if the event should be placed in the queue or *False* if it should not be placed in the queue.

To initialize the serial number component of the event, call *_XSetLastRequestRead* with the event and use the return value.

```
unsigned long _XSetLastRequestRead(display, rep)
      Display *display;
      xGenericReply *rep;
```

display   Specifies the connection to the X server.

rep   Specifies the wire event structure.

This function computes and returns a complete serial number from the partial serial number in the event.

```
Status (*XESetEventToWire(display, event_number, proc))()
      Display *display;
      int event_number;
      int (*proc)();
```

display   Specifies the connection to the X server.

event_number   Specifies the event code.

proc   Specifies the routine to call when converting an event.

You use this procedure to define a procedure to be called when an event needs to be converted from host format (*XEvent*) to wire format (*xEvent*) form. The event number defines which protocol event number to install a conversion routine for. This procedure returns any previously defined procedure. It returns zero if the conversion fails or non-zero otherwise.

Note: You can replace a core event conversion routine with one of your own, although this is not encouraged. It would, however, allow you to intercept a core event and modify it before being sent to another client.

When Xlib needs to convert an event from host format to wire format, your routine is called with these arguments:

```
(*proc)(display, re, event)
      Display *display;
      XEvent *re;
      xEvent *event;
```

The re argument is a pointer to the host format event, and the event argument is a pointer to where the 32-byte wire event structure should be stored. You should fill in the type with the

type from the *XEvent* structure. All other members then should be copied from the host format to the *xEvent* structure.

```
Bool (*XESetWireToError(display, error_number, proc)()
      Display *display;
      int error_number;
      Bool (*proc)();
```

display　　　　Specifies the connection to the X server.

error_number　　Specifies the error code.

proc　　　　　Specifies the routine to call when an error is received.

This function defines a procedure to be called when an extension error needs to be converted from wire format to host format. The error number defines which protocol error code to install the conversion routine for. This procedure returns any previously defined procedure.

Use this function for extension errors that contain additional error values beyond those in a core X error, when multiple wire errors must be combined into a single Xlib error, or when it is necessary to intercept an X error before it is otherwise examined.

When Xlib needs to convert an error from wire format to host format, the routine is called with these arguments:

```
Bool (*proc)(display, he, we)
      Display *display;
      XErrorEvent *he;
      xError *we;
```

The he argument is a pointer to where the host format error should be stored. The structure pointed at by he is guaranteed to be as large as an *XEvent* structure, and so can be cast to a type larger than an *XErrorEvent*, in order to store additional values. If the error is to be completely ignored by Xlib (for example, several protocol error structures will be combined into one Xlib error), then the function should return *False*; otherwise it should return *True*.

```
int (*XESetError(display, extension, proc))()
      Display *display;
      int extension;
      int (*proc)();
```

display　　　　Specifies the connection to the X server.

extension　　　Specifies the extension number.

proc　　　　　Specifies the routine to call when an error is received.

Inside Xlib, there are times that you may want to suppress the calling of the external error handling when an error occurs. This allows status to be returned on a call at the cost of the call being synchronous (though most such routines are query operations, in any case, and are typically programmed to be synchronous).

When Xlib detects a protocol error in *_XReply*, it calls your procedure with these arguments:

```
int (*proc)(display, err, codes, ret_code)
      Display *display;
      xError *err;
      XExtCodes *codes;
      int *ret_code;
```

The err argument is a pointer to the 32-byte wire format error. The codes argument is a pointer to the extension codes structure. The ret_code argument is the return code you may want *_XReply* returned to.

If your routine returns a zero value, the error is not suppressed, and the client's error handler is called. (For further information, see Section 12.8.2) If your routine returns non-zero, the error is suppressed, and *_XReply* returns the value of ret_code.

```
char  *(*XESetErrorString(display, extension, proc))()
      Display *display;
      int extension;
      char *(*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call to obtain an error string.

The *XGetErrorText* function returns a string to the user for an error. *XESetErrorString* allows you to define a routine to be called that should return a pointer to the error message. The following is an example.

```
(*proc)(display, code, codes, buffer, nbytes)
      Display *display;
      int code;
      XExtCodes *codes;
      char *buffer;
      int nbytes;
```

Your procedure is called with the error code for every error detected. You should copy nbytes of a null-terminated string containing the error message into buffer.

```
void (*XESetPrintErrorValues(display, extension, proc))()
      Display *display;
      int extension;
      void (*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when an error is printed.

This function defines a procedure to be called when an extension error is printed, to print the error values. Use this function for extension errors that contain additional error values beyond those in a core X error. This function returns any previously defined procedure.

When Xlib needs to print an error, the routine is called with these arguments:

```
void (*proc)(display, ev, fp)
      Display *display;
      XErrorEvent *ev;
      void *fp;
```

The structure pointed at by ev is guaranteed to be as large as an *XEvent* structure, and so can be cast to a type larger than an *XErrorEvent*, in order to obtain additional values set by using *XESetWireToError*. The underlying type of the fp argument is system-dependent; on a POSIX-compliant fp should be cast to type FILE*.

```
int (*XESetFlushGC(display, extension, proc))()
      Display *display;
      int extension;
      int *(*proc)();
```

display   Specifies the connection to the X server.

extension   Specifies the extension number.

proc   Specifies the routine to call when a GC is flushed.

The *XESetFlushGC* procedure is identical to *XESetCopyGC* except that *XESetFlushGC* is called when a GC cache needs to be updated in the server.

Hooks onto Xlib Data Structures

Various Xlib data structures have provisions for extension routines to chain extension supplied data onto a list. These structures are *GC*, *Visual*, *Screen*, *ScreenFormat*, *Display* and *XFontStruct*. Because the list pointer is always the first member in the structure, a single set of routines can be used to manipulate the data on these lists.

The following structure is used in the routines in this section and is defined in <**X11/Xlib.h**>:

```
typedef struct _XExtData {
      int number;             /* number returned by XInitExtension */
      struct _XExtData *next; /* next item on list of data for structure */
      int (*free_private)();  /* if defined,  called to free private */
      XPointer private_data;  /* data private to this extension. */
} XExtData;
```

When any of the data structures listed above are freed, the list is walked, and the structure's free routine (if any) is called. If free is NULL, then the library frees both the data pointed to by the private_data member and the structure itself.

```
union { Display *display;
      GC gc;
      Visual *visual;
      Screen *screen;
      ScreenFormat *pixmap_format;
      XFontStruct *font } XEDataObject;

XExtData **XEHeadOfExtensionList(object)
      XEDataObject object;
```

object   Specifies the object.

XEHeadOfExtensionList returns a pointer to the list of extension structures attached to the specified object. In concert with *XAddToExtensionList*, *XEHeadOfExtensionList* allows an extension to attach arbitrary data to any of the structures of types contained in *XEDataObject*.

```
XAddToExtensionList(structure, ext_data)
      XExtData **structure;
      XExtData *ext_data;
```

structure   Specifies the extension list.

ext_data   Specifies the extension data structure to add.

The structure argument is a pointer to one of the data structures enumerated above. You must initialize ext_data->number with the extension number before calling this routine.

```
XExtData *XFindOnExtensionList(structure, number)
      struct _XExtData **structure;
      int number;
```

structure   Specifies the extension list.

number   Specifies the extension number from XInitExtension.

XFindOnExtensionList returns the first extension data structure for the extension numbered number. It is expected that an extension will add at most one extension data structure to any single data structure's extension data list. There is no way to find additional structures.

The *XAllocID* macro, which allocates and returns a resource ID, is defined in <**X11/Xlib.h**>.

```
XAllocID(display)
      Display *display;
```

display   Specifies the connection to the X server.

This macro is a call through the *Display* structure to the internal resource ID allocator. It returns a resource ID that you can use when creating new resources.

GC Caching

GCs are cached by the library to allow merging of independent change requests to the same GC into single protocol requests. This is typically called a write-back cache. Any extension routine whose behaviour depends on the contents of a GC must flush the GC cache to make sure the server has up-to-date contents in its GC.

The *FlushGC* macro checks the dirty bits in the library's GC structure and calls *_XFlushGCCache* if any elements have changed. The *FlushGC* macro is defined as follows:

```
FlushGC(display, gc)
      Display *display;
      GC gc;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

Note that if you extend the GC to add additional resource ID components, you should ensure that the library stub sends the change request immediately. This is because a client can free a resource immediately after using it, so if you only stored the value in the cache without forcing a protocol request, the resource might be destroyed before being set into the GC. You can use the *_XFlushGCCache* procedure to force the cache to be flushed. The *_XFlushGCCache* procedure is defined as follows:

```
_XFlushGCCache(display, gc)
      Display *display;
      GC gc;
```

display   Specifies the connection to the X server.

gc   Specifies the GC.

Graphics Batching

If you extend X to add more poly graphics primitives, you may be able to take advantage of facilities in the library to allow back-to-back single calls to be transformed into poly requests. This may dramatically improve performance of programs that are not written using poly requests. A pointer to an *xReq*, called last_req in the display structure, is the last request being processed. By checking that the last request type, drawable, gc and other options are the same as the new one and that there is enough space left in the buffer, you may be able to just extend the previous graphics request by extending the length field of the request and appending the data to the buffer. This can improve performance by five times or more in naive programs. For example, here is the source for the *XDrawPoint* stub. (Writing extension stubs is discussed in the next section.)

```
#include "copyright.h"

#include "Xlibint.h"

/* precompute the maximum size of batching request allowed */

static int size = sizeof(xPolyPointReq) + EPERBATCH * sizeof(xPoint);

XDrawPoint(dpy, d, gc, x, y)
      register Display *dpy;
      Drawable d;
      GC gc;
      int x, y; /* INT16 */
{
      xPoint *point;
      LockDisplay(dpy);
      FlushGC(dpy, gc);
      {
      register xPolyPointReq *req = (xPolyPointReq *) dpy->last_req;
      /* if same as previous request, with same drawable,
         batch requests */
      if (
            (req->reqType == X_PolyPoint)
        && (req->drawable == d)
        && (req->gc == gc->gid)
        && (req->coordMode == CoordModeOrigin)
        && ((dpy->bufptr + sizeof (xPoint)) <= dpy->bufmax)
        && (((char *)dpy->bufptr - (char *)req) < size) ) {
            point = (xPoint *) dpy->bufptr;
            req->length += sizeof (xPoint) >> 2;
            dpy->bufptr += sizeof (xPoint);
            }

      else {
            GetReqExtra(PolyPoint, 4, req); /* 1 point = 4 bytes */
            req->drawable = d;
            req->gc = gc->gid;
            req->coordMode = CoordModeOrigin;
            point = (xPoint *) (req + 1);
            }
      oint->x = x;
      oint->y = y;

      UnlockDisplay(dpy);
      SyncHandle();
}
```

To keep clients from generating very long requests that may monopolize the server, there is a symbol defined in **<X11/Xlibint.h>** of EPERBATCH on the number of requests batched. Most of the performance benefit occurs in the first few merged requests. Note that *FlushGC* is called *before* picking up the value of last_req, because it may modify this field.

Writing Extension Stubs

All X requests always contain the length of the request, expressed as a 16-bit quantity of 32 bits. This means that a single request can be no more than 256K bytes in length. Some servers may not support single requests of such a length. The value of dpy->max_request_size contains the maximum length as defined by the server implementation. For further information, see the **X Window System Protocol** specification.

Requests, Replies and Xproto.h

The **<X11/Xproto.h>** file contains three sets of definitions that are of interest to the stub implementor: request names, request structures and reply structures.

You need to generate a file equivalent to **<X11/Xproto.h>** for your extension and need to include it in your stub routine. Each stub routine also must include **<X11/Xlibint.h>**.

The identifiers are deliberately chosen in such a way that, if the request is called X_DoSomething, then its request structure is xDoSomethingReq, and its reply is xDoSomethingReply. The GetReq family of macros, defined in **<X11/Xlibint.h>**, takes advantage of this naming scheme.

For each X request, there is a definition in **<X11/Xproto.h>** that looks similar to this:

```
#define X_DoSomething    42
```

In your extension header file, this will be a minor opcode, instead of a major opcode.

Request Format

Every request contains an 8-bit major opcode and a 16-bit length field expressed in units of four bytes. Every request consists of four bytes of header (containing the major opcode, the length field and a data byte) followed by zero or more additional bytes of data. The length field defines the total length of the request, including the header. The length field in a request must equal the minimum length required to contain the request. If the specified length is smaller or larger than the required length, the server should generate a *BadLength* error. Unused bytes in a request are not required to be zero. Extensions should be designed in such a way that long protocol requests can be split up into smaller requests, if it is possible to exceed the maximum request size of the server. The protocol guarantees the maximum request size to be no smaller than 4096 units (16384 bytes).

Major opcodes 128 through 255 are reserved for extensions. Extensions are intended to contain multiple requests, so extension requests typically have an additional minor opcode encoded in the ''spare'' data byte in the request header, but the placement and interpretation of this minor opcode as well as all other fields in extension requests are not defined by the core protocol. Every request is implicitly assigned a sequence number (starting with one) used in replies, errors and events.

To help but not cure portability problems to certain machines, the *B16* and *B32* macros have been defined so that they can become bitfield specifications on some machines. For example, on a Cray, these should be used for all 16-bit and 32-bit quantities, as discussed below.

Most protocol requests have a corresponding structure typedef in **<X11/Xproto.h>**, which looks like:

```
typedef struct _DoSomethingReq {
      CARD8 reqType;      /* X_DoSomething */
      CARD8 someDatum;    /* used differently in different requests */
      CARD16 length B16;  /* total # of bytes in request, divided by 4 */
      ...
      /* request-specific data */
      ...
} xDoSomethingReq;
```

If a core protocol request has a single 32-bit argument, you need not declare a request structure in your extension header file. Instead, such requests use **<X11/Xproto.h>**'s *xResourceReq* structure. This structure is used for any request whose single argument is a *Window*, *Pixmap*, *Drawable*, *GContext*, *Font*, *Cursor*, *Colormap*, *Atom* or *VisualID*.

```
typedef struct _ResourceReq {
      CARD8 reqType;      /* the request type, e.g. X_DoSomething */
      BYTE pad;           /* not used */
      CARD16 length B16;  /* 2 (= total # of bytes in request,
                             divided by 4) */
      CARD32 id B32;      /* the Window, Drawable, Font, GContext, etc. */
} xResourceReq;
```

If convenient, you can do something similar in your extension header file.

In both of these structures, the reqType field identifies the type of the request (for example, X_MapWindow or X_CreatePixmap). The length field tells how long the request is in units of 4-byte longwords. This length includes both the request structure itself and any variable-length data, such as strings or lists, that follow the request structure. Request structures come in different sizes, but all requests are padded to be multiples of four bytes long.

A few protocol requests take no arguments at all. Instead, they use **<X11/Xproto.h>**'s *xReq* structure, which contains only a reqType and a length (and a pad byte).

If the protocol request requires a reply, then **<X11/Xproto.h>** also contains a reply structure typedef:

```
typedef struct _DoSomethingReply {
      BYTE type;                  /* always X_Reply */
      BYTE someDatum;             /* used differently in different requests */
      CARD16 sequenceNumber B16;  /* # of requests sent so far */
      CARD32 length B32;          /* # of additional bytes, divided by 4 */
      ...
      /* request-specific data */
      ...
} xDoSomethingReply;
```

Most of these reply structures are 32 bytes long. If there are not that many reply values, then they contain a sufficient number of pad fields to bring them up to 32 bytes. The length field is the total number of bytes in the request minus 32, divided by 4. This length will be non-zero only if:

• The reply structure is followed by variable-length data such as a list or string.

• The reply structure is longer than 32 bytes.

Only *GetWindowAttributes*, *QueryFont*, *QueryKeymap* and *GetKeyboardControl* have reply structures longer than 32 bytes in the core protocol.

A few protocol requests return replies that contain no data. **<X11/Xproto.h>** does not define reply structures for these. Instead, they use the *xGenericReply* structure, which contains only a type, length and sequence number (and sufficient padding to make it 32 bytes long).

Starting to Write a Stub Routine

An Xlib stub routine should always start like this:

```
#include "Xlibint.h"

XDoSomething (arguments, ... )
      /* argument declarations */
      {

      register XDoSomethingReq *req;
      ...
```

If the protocol request has a reply, then the variable declarations should include the reply structure for the request. The following is an example:

```
xDoSomethingReply rep;
```

Locking Data Structures

To lock the display structure for systems that want to support multithreaded access to a single display connection, each stub will need to lock its critical section. Generally, this section is the point from just before the appropriate GetReq call until all arguments to the call have been stored into the buffer. The precise instructions needed for this locking depend upon the machine architecture. Two calls, which are generally implemented as macros, have been provided.

```
LockDisplay(display)
      Display *display;

UnlockDisplay(display)
      Display *display;
```

display   Specifies the connection to the X server.

Sending the Protocol Request and Arguments

After the variable declarations, a stub routine should call one of four macros defined in **<X11/Xlibint.h>**: *GetReq*, *GetReqExtra*, *GetResReq* or *GetEmptyReq*. All of these macros take, as their first argument, the name of the protocol request as declared in **<X11/Xproto.h>** except with X_ removed. Each one declares a *Display* structure pointer, called dpy, and a pointer to a request structure, called req, which is of the appropriate type. The macro then appends the request structure to the output buffer, fills in its type and length field, and sets req to point to it.

If the protocol request has no arguments (for instance, X_GrabServer), then use *GetEmptyReq*.

```
GetEmptyReq (DoSomething, req);
```

If the protocol request has a single 32-bit argument (such as a *Pixmap*, *Window*, *Drawable*, *Atom*, and so on), then use *GetResReq*. The second argument to the macro is the 32-bit object. *X_MapWindow* is a good example.

```
GetResReq (DoSomething, rid, req);
```

The rid argument is the *Pixmap*, *Window* or other resource ID.

If the protocol request takes any other argument list, then call *GetReq*. After the *GetReq*, you need to set all the other fields in the request structure, usually from arguments to the stub routine.

```
GetReq (DoSomething, req);
/* fill in arguments here */
req->arg1 = arg1;
req->arg2 = arg2;
...
```

A few stub routines (such as *XCreateGC* and *XCreatePixmap*) return a resource ID to the caller but pass a resource ID as an argument to the protocol request. Such routines use the macro *XAllocID* to allocate a resource ID from the range of IDs that were assigned to this client when it opened the connection.

```
rid = req->rid = XAllocID();
...
return (rid);
```

Finally, some stub routines transmit a fixed amount of variable-length data after the request. Typically, these routines (such as *XMoveWindow* and *XSetBackground*) are special cases of more general functions like *XMoveResizeWindow* and *XChangeGC*. These special case routines use *GetReqExtra*, which is the same as *GetReq* except that it takes an additional argument (the number of extra bytes to allocate in the output buffer after the request structure). This number should always be a multiple of four.

Variable-length Arguments

Some protocol requests take additional variable-length data that follow the *xDoSomethingReq* structure. The format of this data varies from request to request. Some requests require a sequence of 8-bit bytes, others a sequence of 16-bit or 32-bit entities, and still others a sequence of structures.

It is necessary to add the length of any variable-length data to the length field of the request structure. That length field is in units of 32-bit longwords. If the data is a string or other sequence of 8-bit bytes, then you must round the length up and shift it before adding:

```
req->length += (nbytes+3)>>2;
```

To transmit variable-length data, use the *Data* macros. If the data fits into the output buffer, then this macro copies it to the buffer. If it does not fit, however, the *Data* macro calls *_XSend*, which transmits first the contents of the buffer and then your data. The *Data* macros take three arguments: the Display, a pointer to the beginning of the data, and the number of bytes to be sent.

```
Data(display, (char *) data, nbytes);

Data16(display, (short *) data, nbytes);

Data26(display, (long *) data, nbytes);
```

Data, *Data16* and *Data32* are macros that may use their last argument more than once, so that argument should be a variable rather than an expression such as ''nitems*sizeof(item)''. You should do that kind of computation in a separate statement before calling them. Use the appropriate macro when sending byte, short or long data.

If the protocol request requires a reply, then call the procedure *_XSend* instead of the *Data* macro. *_XSend* takes the same arguments, but because it sends your data immediately instead of

copying it into the output buffer (which would later be flushed anyway by the following call on *_XReply*), it is faster.

Replies

If the protocol request has a reply, then call *_XReply* after you have finished dealing with all the fixed and variable-length arguments. *_XReply* flushes the output buffer and waits for an *xReply* packet to arrive. If any events arrive in the meantime, *_XReply* places them in the queue for later use.

```
Status _XReply(display, rep, extra, discard)
      Display *display;
      xReply *rep;
      int extra;
      Bool discard;
```

display   Specifies the connection to the X server.

rep   Specifies the reply structure.

extra   Specifies the number of 32-bit words expected after the replay.

discard   Specifies if beyond the ''extra'' data should be discarded.

_XReply waits for a reply packet and copies its contents into the specified rep. *_XReply* handles error and event packets that occur before the reply is received. *_XReply* takes four arguments:

- a *Display* * structure

- a pointer to a reply structure (which must be cast to an *xReply* *)

- the number of additional 32-bit words (beyond *sizeof(*xReply*)* = 32 bytes) in the reply structure

- a Boolean that indicates whether *_XReply* is to discard any additional bytes beyond those it was told to read.

Because most reply structures are 32 bytes long, the third argument is usually 0. The only core protocol exceptions are the replies to *GetWindowAttributes*, *QueryFont*, *QueryKeymap* and *GetKeyboardControl*, which have longer replies.

The last argument should be *False* if the reply structure is followed by additional variable-length data (such as a list or string). It should be *True* if there is not any variable-length data.

Note: This last argument is provided for upward-compatibility reasons to allow a client to communicate properly with a hypothetical later version of the server that sends more data than the client expected. For example, some later version of *GetWindowAttributes* might use a larger, but compatible, *xGetWindowAttributesReply* that contains additional attribute data at the end.

_XReply returns *True* if it received a reply successfully or *False* if it received any sort of error.

For a request with a reply that is not followed by variable-length data, you write something like:

```
_XReply(display, (xReply *)&rep, 0, True);
        *ret1 = rep.ret1;
        *ret2 = rep.ret2;
        *ret3 = rep.ret3;
        ...
        UnlockDisplay(dpy);
        SyncHandle();
        return (rep.ret4);
}
```

If there is variable-length data after the reply, change the *True* to *False*, and use the appropriate *_XRead* function to read the variable-length data.

```
_XRead(display, data_return, nbytes)
        Display *display;
        char *data_return;
        long nbytes;
```

display   Specifies the connection to the X server.

data_return   Specifies the buffer.

nbytes   Specifies the number of bytes required.

_XRead reads the specified number of bytes into data_return.

```
_XRead16(display, data_return, nbytes)
        Display *display;
        short *data_return;
        long nbytes;
```

display   Specifies the connection to the X server.

data_return   Specifies the buffer.

nbytes   Specifies the number of bytes required.

_XRead16 reads the specified number of bytes, unpacking them as 16-bit quantities, into the specified array as shorts.

```
_XRead32(display, data_return, nbytes)
        Display *display;
        long *data_return;
        long nbytes;
```

display   Specifies the connection to the X server.

data_return   Specifies the buffer.

nbytes   Specifies the number of bytes required.

_XRead32 reads the specified number of bytes, unpacking them as 32-bit quantities, into the specified array as longs.

```
_XRead16Pad(display, data_return, nbytes)
        Display *display;
        short *data_return;
        long nbytes;
```

display   Specifies the connection to the X server.

data_return   Specifies the buffer.

nbytes   Specifies the number of bytes required.

_XRead16Pad reads the specified number of bytes, unpacking them as 16-bit quantities, into the specified array as shorts. If the number of bytes is not a multiple of four, *_XRead16Pad* reads and discards up to three additional pad bytes.

```
_XReadPad(display, data_return, nbytes)
      Display *display;
      char *data_return;
      long nbytes;
```

display   Specifies the connection to the X server.

data_return   Specifies the buffer.

nbytes   Specifies the number of bytes required.

_XReadPad reads the specified number of bytes into data_return. If the number of bytes is not a multiple of four, *_XReadPad* reads and discards up to three additional pad bytes.

Each protocol request is a little different. For further information, see the Xlib sources for examples.

Synchronous Calling

To ease debugging, each routine should have a call, just before returning to the user, to a routine called *SyncHandle*. This routine generally is implemented as a macro. If synchronous mode is enabled (see *XSynchronize*), the request is sent immediately. The library, however, waits until any error the routine could generate at the server has been handled.

Allocating and Deallocating Memory

To support the possible reentry of these routines, you must observe several conventions when allocating and deallocating memory, most often done when returning data to the user from the window system of a size the caller could not know in advance (for example, a list of fonts or a list of extensions). The standard C library routines on many systems are not protected against signals or other multithreaded uses. The following analogies to standard I/O library routines have been defined:

Xmalloc( )   Replaces *malloc*( ).

XFree( )   Replaces *free*( ).

Xcalloc( )   Replaces *calloc*( ).

These should be used in place of any calls you would make to the normal C library routines.

If you need a single scratch buffer inside a critical section (for example, to pack and unpack data to and from the wire protocol), the general memory allocators may be too expensive to use (particularly in output routines, which are performance critical). The routine below returns a scratch buffer for your use:

```
char *_XAllocScratch(display, nbytes)
      Display *display;
      unsigned long nbytes;
```

display   Specifies the connection to the X server.

nbytes   Specifies the number of bytes required.

This storage must only be used inside of the critical section of your stub.

Portability Considerations

Many machine architectures, including many of the more recent RISC architectures, do not correctly access data at unaligned locations; their compilers pad out structures to preserve this characteristic. Many other machines capable of unaligned references pad inside of structures as well to preserve alignment, because accessing aligned data is usually much faster. Because the library and the server use structures to access data at arbitrary points in a byte stream, all data in request and reply packets *must* be naturally aligned; that is, 16-bit data starts on 16-bit boundaries in the request and 32-bit data on 32-bit boundaries. All requests *must* be a multiple of 32 bits in length to preserve the natural alignment in the data stream. You must pad structures out to 32-bit boundaries. Pad information does not have to be zeroed unless you want to preserve such fields for future use in your protocol requests. Floating point varies radically between machines and should be avoided completely if at all possible.

This code may run on machines with 16-bit ints. So, if any integer argument, variable, or return value either can take only non-negative values or is declared as a CARD16 in the protocol, be sure to declare it as unsigned int and not as int. (This, of course, does not apply to Booleans or enumerations.)

Similarly, if any integer argument or return value is declared CARD32 in the protocol, declare it as an unsigned long and not as int or long. This also goes for any internal variables that may take on values larger than the maximum 16-bit unsigned int.

The library currently assumes that a char is 8 bits, a short is 16 bits, an int is 16 or 32 bits, and a long is 32 bits. The *PackData* macro is a half-hearted attempt to deal with the possibility of 32-bit shorts. However, much more work is needed to make this work properly.

Deriving the Correct Extension Opcode

The remaining problem a writer of an extension stub routine faces that the core protocol does not face is to map from the call to the proper major and minor opcodes. While there are a number of strategies, the simplest and fastest is outlined below.

1. Declare an array of pointers, _NFILE long (this is normally found in **<stdio.h>** and is the number of file descriptors supported on the system) of type *XExtCodes*. Make sure these are all initialized to NULL.

2. When your stub is entered, your initialization test is just to use the display pointer passed in to access the file descriptor and an index into the array. If the entry is NULL, then this is the first time you are entering the routine for this display. Call your initialization routine and pass it to the display pointer.

3. Once in your initialization routine, call *XInitExtension*; if it succeeds, store the pointer returned into this array. Make sure to establish a close display handler to allow you to zero the entry. Do whatever other initialization your extension requires. (For example, install event handlers and so on.) Your initialization routine would normally return a pointer to the *XExtCodes* structure for this extension, which is what would normally be found in your array of pointers.

4. After returning from your initialization routine, the stub can now continue normally, because it has its major opcode safely in its hand in the *XExtCodes* structure.

Appendix D

Compatibility Functions

The X Version 11 and X Version 10 functions discussed in this appendix are obsolete, have been superseded by newer X Version 11 functions, and are maintained for compatibility reasons only.

X Version 11 Compatibility Functions

X Version 11 compatibility functions can be used to:

- set standard properties
- set and get window sizing hints
- set and get an XStandardColormap structure
- parse window geometry
- get X environment defaults.

Setting Standard Properties

To specify a minimum set of properties describing the ''quickie'' application, use *XSetStandardProperties*. This function has been superseded by *XSetWMProperties* and sets all or portions of the WM_NAME, WM_ICON_NAME, WM_HINTS, WM_COMMAND and WM_NORMAL_HINTS properties.

```
XSetStandardProperties(display, w, window_name,
        icon_name, icon_pixmap, argv, argc, hints)
    Display *display;
    Window w;
    char *window_name;
    char *icon_name;
    Pixmap icon_pixmap;
    char **argv;
    int argc;
    XSizeHints *hints;
```

display   Specifies the connection to the X server.

w   Specifies the window.

window_name   Specifies the window name, which should be a null-terminated string.

icon_name   Specifies the icon name, which should be a null-terminated string.

icon_pixmap   Specifies the bitmap that is to be used for the icon or *None*.

argv   Specifies the application's argument list.

argc   Specifies the number of arguments.

hints   Specifies a pointer to the size hints for the window in its normal state.

The *XSetStandardProperties* function provides a means by which simple applications set the most essential properties with a single call. *XSetStandardProperties* should be used to give a window manager some information about your program's preferences. It should not be used by applications that need to communicate more information than is possible with *XSetStandardProperties*. (Typically, argv is the argv array of your main program.) If the strings

are not in the Host Portable Character Encoding the result is implementation-dependent.

XSetStandardProperties can generate *BadAlloc* and *BadWindow* errors.

Setting and Getting Window Sizing Hints

Xlib provides functions that you can use to set or get window sizing hints. The functions discussed in this section use the flags and the *XSizeHints* structure, as defined in the **<X11/Xutil.h>** header file, and use the WM_NORMAL_HINTS property.

To set the size hints for a given window in its normal state, use *XSetNormalHints*. This function has been superseded by *XSetWMNormalHints*.

```
XSetNormalHints(display, w, hints)
      Display *display;
      Window w;
      XSizeHints *hints;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints   Specifies a pointer to the size hints for the window in its normal state.

The *XSetNormalHints* function sets the size hints structure for the specified window. Applications use *XSetNormalHints* to inform the window manager of the size or position desirable for that window. In addition, an application that wants to move or resize itself should call *XSetNormalHints* and specify its new desired location and size as well as making direct Xlib calls to move or resize. This is because window managers may ignore redirected configure requests, but they pay attention to property changes.

To set size hints, an application not only must assign values to the appropriate members in the hints structure but also must set the flags member of the structure to indicate which information is present and where it came from. A call to *XSetNormalHints* is meaningless, unless the flags member is set to indicate which members of the structure have been assigned values.

XSetNormalHints can generate *BadAlloc* and *BadWindow* errors.

To return the size hints for a window in its normal state, use *XGetNormalHints*. This function has been superseded by *XGetWMNormalHints*.

```
Status XGetNormalHints(display, w, hints_return)
      Display *display;
      Window w;
      XSizeHints *hints_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints_return   Returns the size hints for the window in its normal state.

The *XGetNormalHints* function returns the size hints for a window in its normal state. It returns a non-zero status if it succeeds or zero if the application specified no normal size hints for this window.

XGetNormalHints can generate a *BadWindow* error.

The next two functions set and read the WM_ZOOM_HINTS property.

To set the zoom hints for a window, use *XSetZoomHints*. This function is no longer supported by the **ICCCM** specification.

```
XSetZoomHints(display, w, zhints)
      Display *display;
      Window w;
      XSizeHints *zhints;
```

display   Specifies the connection to the X server.

w   Specifies the window.

zhints   Specifies a pointer to the zoom hints.

Many window managers think of windows in one of three states: iconic, normal or zoomed. The *XSetZoomHints* function provides the window manager with information for the window in the zoomed state.

XSetZoomHints can generate *BadAlloc* and *BadWindow* errors.

To read the zoom hints for a window, use *XGetZoomHints*. This function is no longer supported by the **ICCCM** specification.

```
Status XGetZoomHints(display, w, zhints_return)
      Display *display;
      Window w;
      XSizeHints *zhints_return;
```

display   Specifies the connection to the X server.

w   Specifies the window.

zhints_return   Returns the zoom hints.

The *XGetZoomHints* function returns the size hints for a window in its zoomed state. It returns a non-zero status if it succeeds or zero if the application specified no zoom size hints for this window.

XGetZoomHints can generate a *BadWindow* error.

To set the value of any property of type WM_SIZE_HINTS, use *XSetSizeHints*. This function has been superseded by *XSetWMSizeHints*.

```
XSetSizeHints(display, w, hints, property)
      Display *display;
      Window w;
      XSizeHints *hints;
      Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints   Specifies a pointer to the size hints.

property   Specifies the property name.

The *XSetSizeHints* function sets the *XSizeHints* structure for the named property and the specified window. This is used by *XSetNormalHints* and *XSetZoomHints*, and can be used to set the value of any property of type WM_SIZE_HINTS. Thus, it may be useful if other properties of that type get defined.

XSetSizeHints can generate *BadAlloc*, *BadAtom* and *BadWindow* errors.

To read the value of any property of type WM_SIZE_HINTS, use *XGetSizeHints*. This function has been superseded by *XGetWMSizeHints*.

```
Status XGetSizeHints(display, w, hints_return, property)
      Display *display;
      Window w;
      XSizeHints *hints_return;
      Atom property;
```

display   Specifies the connection to the X server.

w   Specifies the window.

hints_return   Returns the size hints.

property   Specifies the property name.

XGetSizeHints returns the *XSizeHints* structure for the named property and the specified window. This is used by *XGetNormalHints* and *XGetZoomHints*. It also can be used to retrieve the value of any property of type WM_SIZE_HINTS. Thus, it may be useful if other properties of that type get defined. *XGetSizeHints* returns a non-zero status if a size hint was defined or zero otherwise.

XGetSizeHints can generate *BadAtom* and *BadWindow* errors.

Getting and Setting an XStandardColormap Structure

To get the *XStandardColormap* structure associated with one of the described atoms, use *XGetStandardColormap*. This function has been superseded by *XGetRGBColormap*.

```
Status XGetStandardColormap(display, w, colormap_return, property)
      Display *display;
      Window w;
      XStandardColormap *colormap_return;
      Atom property; /* RGB_BEST_MAP, etc. */
```

display   Specifies the connection to the X server.

w   Specifies the window.

colormap_return   Returns the colormap associated with the specified atom.

property   Specifies the property name.

The *XGetStandardColormap* function returns the colormap definition associated with the atom supplied as the property argument. *XGetStandardColormap* returns a non-zero status if successful, and zero otherwise. For example, to fetch the standard *GrayScale* colormap for a display, you use *XGetStandardColormap* with the following syntax:

```
XGetStandardColormap(dpy, DefaultRootWindow(dpy), &cmap, XA_RGB_GRAY_MAP);
```

See Section 15.3 for the semantics of standard colormaps.

XGetStandardColormap can generate *BadAtom* and *BadWindow* errors.

To set a standard colormap, use *XSetStandardColormap*. This function has been superseded by *XSetRGBColormap*.

```
XSetStandardColormap(display, w, colormap, property)
      Display *display;
      Window w;
      XStandardColormap *colormap;
      Atom property; /* RGB_BEST_MAP, etc. */
```

display	Specifies the connection to the X server.
w	Specifies the window.
colormap	Specifies the colormap.
property	Specifies the property name.

The *XSetStandardColormap* function usually is only used by window or session managers.

XSetStandardColormap can generate *BadAlloc*, *BadAtom*, *BadDrawable* and *BadWindow* errors.

Parsing Window Geometry

To parse window geometry given a user-specified position and a default position, use *XGeometry*. This function has been superseded by *XWMGeometry*.

```
int XGeometry(display, screen, position, default_position,
          bwidth, fwidth, fheight, xadder, yadder,
          x_return, y_return, width_return, height_return)
     Display *display;
     int screen;
     char *position, *default_position;
     unsigned int bwidth;
     unsigned int fwidth, fheight;
     int xadder, yadder;
     int *x_return, *y_return;
     int *width_return, *height_return;
```

display	Specifies the connection to the X server.
screen	Specifies the screen.
position *default_position*	Specify the geometry specifications.
bwidth	Specifies the border width.
fheight *fwidth*	Specify the font height and width in pixels (increment size).
xadder *yadder*	Specify additional interior padding needed in the window.
x_return *y_return*	Return the x and y offsets.
width_return *height_return*	Return the width and height determined.

You pass in the border width (bwidth), size of the increments fwidth and fheight (typically font width and height), and any additional interior space (xadder and yadder) to make it easy to compute the resulting size. The *XGeometry* function returns the position the window should be placed given a position and a default position. *XGeometry* determines the placement of a window using a geometry specification as specified by *XParseGeometry* and the additional information about the window. Given a fully qualified default geometry specification and an incomplete geometry specification, *XParseGeometry* returns a bitmask value as defined above in the *XParseGeometry* call, by using the position argument.

The returned width and height will be the width and height specified by default_position as overridden by any user-specified position. They are not affected by fwidth, fheight, xadder or

yadder. The x and y coordinates are computed by using the border width, the screen width and height, padding as specified by xadder and yadder, and the fheight and fwidth times the width and height from the geometry specifications.

Obtaining the X Environment Defaults

The *XGetDefault* function provides a primitive interface to the resource manager facilities discussed in Chapter 16. It is only useful in very simple applications.

```
char *XGetDefault(display, program, option)
      Display *display;
      char *program;
      char *option;
```

display   Specifies the connection to the X server.

program   Specifies the program name for the Xlib defaults (usually argv[0] of the main program).

option   Specifies the option name.

The *XGetDefault* function returns the value of the resource *prog.option*, where *prog* is the program argument with the directory prefix removed and *option* must be a single component. Note that multi-level resources cannot be used with *XGetDefault*. The class ''Program.Name'' is always used for the resource lookup. If the specified option name does not exist for this program, *XGetDefault* returns NULL. The strings returned by *XGetDefault* are owned by Xlib and should not be modified or freed by the client.

If a database has been set with *XrmSetDatabase*, that database is used for the lookup. Otherwise, a database is created as described below, and this is set in the display (as if by calling *XrmSetDatabase*). The database is created in the current locale. To create a database, *XGetDefault* uses resources from the RESOURCE_MANAGER property on the root window of screen zero. If no such property exists, a resource file in the user's home directory is used. On a POSIX-conformant system, this file is *$HOME/.Xdefaults*. After loading these defaults, *XGetDefault* merges additional defaults specified by the XENVIRONMENT environment variable. If XENVIRONMENT is defined, it contains a full path name for the additional resource file. If XENVIRONMENT is not defined, *XGetDefault* looks for *$HOME/.Xdefaults-name,* where *name* specifies the name of the machine on which the application is running.

X Version 10 Compatibility Functions

You can use the X Version 10 compatibility functions to:

- draw and fill polygons and curves
- associate user data with a value.

Drawing and Filling Polygons and Curves

Xlib provides functions that you can use to draw or fill arbitrary polygons or curves. These functions are provided mainly for compatibility with X Version 10 and have no server support. That is, they call other Xlib functions, not the server directly. Thus, if you just have straight lines to draw, using *XDrawLines* or *XDrawSegments* is much faster.

The functions discussed here provide all the functionality of the X Version 10 functions *XDraw*, *XDrawFilled*, *XDrawPatterned*, *XDrawDashed* and *XDrawTiled*. They are as compatible as possible given X Version 11's new line drawing functions. One thing to note, however, is that *VertexDrawLastPoint* is no longer supported. Also, the error status returned is the opposite of

what it was under X Version 10 (this is the X Version 11 standard error status). *XAppendVertex* and *XClearVertexFlag* from X Version 10 also are not supported.

Just how the graphics context you use is set up actually determines whether you get dashes or not, and so on. Lines are properly joined if they connect and include the closing of a closed figure (see *XDrawLines*). The functions discussed here fail (return zero) only if they run out of memory or are passed a *Vertex* list that has a *Vertex* with *VertexStartClosed* set that is not followed by a *Vertex* with *VertexEndClosed* set.

To achieve the effects of the X Version 10 *XDraw*, *XDrawDashed*, and *XDrawPatterned*, use *XDraw*.

```
#include <X11/X10.h>

Status XDraw(display, d, gc, vlist, vcount)
      Display *display;
      Drawable d;
      GC gc;
      Vertex *vlist;
      int vcount;
```

display   Specifies the connection to the X server.

d   Specifies the drawable.

gc   Specifies the GC.

vlist   Specifies a pointer to the list of vertices that indicate what to draw.

vcount   Specifies how many vertices are in vlist.

XDraw draws an arbitrary polygon or curve. The figure drawn is defined by the specified list of vertices (vlist). The points are connected by lines as specified in the flags in the vertex structure.

Each Vertex, as defined in <**X11/X10.h**>, is a structure with the following members:

```
typedef struct _Vertex {
      short x,y;
      unsigned short flags;
} Vertex;
```

The x and y members are the coordinates of the vertex that are relative to either the upper-left inside corner of the drawable (if *VertexRelative* is zero) or the previous vertex (if *VertexRelative* is one).

The flags, as defined in <**X11/X10.h**>, are as follows:

```
VertexRelative        0x0001    /* else absolute */
VertexDontDraw        0x0002    /* else draw */
VertexCurved          0x0004    /* else straight */
VertexStartClosed     0x0008    /* else not */
VertexEndClosed       0x0010    /* else not */
```

- If *VertexRelative* is not set, the coordinates are absolute (that is, relative to the drawable's origin). The first vertex must be an absolute vertex.

- If *VertexDontDraw* is one, no line or curve is drawn from the previous vertex to this one. This is analogous to picking up the pen and moving to another place before drawing another line.

- If *VertexCurved* is one, a spline algorithm is used to draw a smooth curve from the previous vertex through this one to the next vertex. Otherwise, a straight line is drawn from the

previous vertex to this one. It makes sense to set *VertexCurved* to one only if a previous and next vertex are both defined (either explicitly in the array or through the definition of a closed curve).

- It is permissible for *VertexDontDraw* bits and *VertexCurved* bits both to be one. This is useful if you want to define the previous point for the smooth curve but do not want an actual curve drawing to start until this point.

- If *VertexStartClosed* is one, then this point marks the beginning of a closed curve. This vertex must be followed later in the array by another vertex whose effective coordinates are identical and that has a *VertexEndClosed* bit of one. The points in between form a cycle to determine predecessor and successor vertices for the spline algorithm.

This function uses these GC components: function, plane-mask, line-width, line-style, cap-style, join-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. It also uses these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, tile-stipple-y-origin, dash-offset and dash-list.

To achieve the effects of the X Version 10 *XDrawTiled* and *XDrawFilled*, use *XDrawFilled*.

```
#include <X11/X10.h>

Status XDrawFilled(display, d, gc, vlist, vcount)
      Display *display;
      Drawable d;
      GC gc;
      Vertex *vlist;
      int vcount;
```

display	Specifies the connection to the X server.
d	Specifies the drawable.
gc	Specifies the GC.
vlist	Specifies a pointer to the list of vertices that indicate what to draw.
vcount	Specifies how many vertices are in vlist.

XDrawFilled draws arbitrary polygons or curves and then fills them.

This function uses these GC components: function, plane-mask, line-width, line-style, cap-style, join-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin and clip-mask. It also uses these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, tile-stipple-y-origin, dash-offset, dash-list, fill-style and fill-rule.

Associating User Data with a Value

These functions have been superseded by the context management functions (see Section 17.10). It is often necessary to associate arbitrary information with resource IDs. Xlib provides the *XAssocTable* functions that you can use to make such an association. Application programs often need to be able to easily refer to their own data structures when an event arrives. The *XAssocTable* system provides users of the X library with a method for associating their own data structures with X resources (*Pixmaps*, *Fonts*, *Windows* and so on).

An *XAssocTable* can be used to type X resources. For example, the user may want to have three or four types of windows, each with different properties. This can be accomplished by associating each X window ID with a pointer to a window property data structure defined by the user. A generic type has been defined in the X library for resource IDs. It is called an XID.

There are a few guidelines that should be observed when using an *XAssocTable*:

- All XIDs are relative to the specified display.

- Because of the hashing scheme used by the association mechanism, the following rules for determining the size of a *XAssocTable* should be followed. Associations will be made and looked up more efficiently if the table size (number of buckets in the hashing system) is a power of two and if there are not more than 8 XIDs per bucket.

To return a pointer to a new *XAssocTable*, use *XCreateAssocTable*.

```
XAssocTable *XCreateAssocTable(size)
      int size;
```

size   Specifies the number of buckets in the hash system of *XAssocTable*.

The size argument specifies the number of buckets in the hash system of *XAssocTable*. For reasons of efficiency the number of buckets should be a power of two. Some size suggestions might be: use 32 buckets per 100 objects, and a reasonable maximum number of objects per buckets is 8. If an error allocating memory for the *XAssocTable* occurs, a NULL pointer is returned.

To create an entry in a given *XAssocTable*, use *XMakeAssoc*.

```
XMakeAssoc(display, table, x_id, data)
      Display *display;
      XAssocTable *table;
      XID x_id;
      char *data;
```

display   Specifies the connection to the X server.

table   Specifies the assoc table.

x_id   Specifies the X resource ID.

data   Specifies the data to be associated with the X resource ID.

XMakeAssoc inserts data into an *XAssocTable* keyed on an XID. Data is inserted into the table only once. Redundant inserts are ignored. The queue in each association bucket is sorted from the lowest XID to the highest XID.

To obtain data from a given *XAssocTable*, use *XLookUpAssoc*.

```
char *XLookUpAssoc(display, table, x_id)
      Display *display;
      XAssocTable *table;
      XID x_id;
```

display   Specifies the connection to the X server.

table   Specifies the assoc table.

x_id   Specifies the X resource ID.

XLookUpAssoc retrieves the data stored in an *XAssocTable* by its XID. If an appropriately matching XID can be found in the table, *XLookUpAssoc* returns the data associated with it. If the x_id cannot be found in the table, it returns NULL.

To delete an entry from a given *XAssocTable*, use *XDeleteAssoc*.

```
XDeleteAssoc(display, table, x_id)
      Display *display;
      XAssocTable *table;
      XID x_id;
```

display   Specifies the connection to the X server.

table   Specifies the assoc table.

x_id   Specifies the X resource ID.

XDeleteAssoc deletes an association in an *XAssocTable* keyed on its XID. Redundant deletes (and deletes of nonexistent XIDs) are ignored. Deleting associations in no way impairs the performance of an *XAssocTable*.

To free the memory associated with a given *XAssocTable*, use *XDestroyAssocTable*.

```
XDestroyAssocTable(table)
      XAssocTable *table;
```

table   Specifies the assoc table.

Index