

Xlib – C Language X Interface
X Window System Standard
X Version 11, Release 6.7

James Gettys
Cambridge Research Laboratory
Digital Equipment Corporation

Robert W. Scheifler
Laboratory for Computer Science
Massachusetts Institute of Technology

with contributions from

Chuck Adams, Tektronix, Inc.

Vania Joloboff, Open Software Foundation

Hideki Hiura, Sun Microsystems, Inc.

Bill McMahon, Hewlett-Packard Company

Ron Newman, Massachusetts Institute of Technology

Al Tabayoyon, Tektronix, Inc.

Glenn Widener, Tektronix, Inc.

Shigeru Yamada, Fujitsu OSSI

The X Window System is a trademark of The Open Group.

TekHVC is a trademark of Tektronix, Inc.

Copyright © 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1994, 1996, 2002 The Open Group

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE X CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of The Open Group shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from The Open Group.

Copyright © 1985, 1986, 1987, 1988, 1989, 1990, 1991 by Digital Equipment Corporation

Portions Copyright © 1990, 1991 by Tektronix, Inc.

Permission to use, copy, modify and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in all copies, and that the names of Digital and Tektronix not be used in in advertising or publicity pertaining to this documentation without specific, written prior permission. Digital and Tektronix makes no representations about the suitability of this documentation for any purpose. It is provided "as is" without express or implied warranty.

Acknowledgments

The design and implementation of the first 10 versions of X were primarily the work of three individuals: Robert Scheifler of the MIT Laboratory for Computer Science and Jim Gettys of Digital Equipment Corporation and Ron Newman of MIT, both at MIT Project Athena. X version 11, however, is the result of the efforts of dozens of individuals at almost as many locations and organizations. At the risk of offending some of the players by exclusion, we would like to acknowledge some of the people who deserve special credit and recognition for their work on Xlib. Our apologies to anyone inadvertently overlooked.

Release 1

Our thanks goes to Ron Newman (MIT Project Athena), who contributed substantially to the design and implementation of the Version 11 Xlib interface.

Our thanks also goes to Ralph Swick (Project Athena and Digital) who kept it all together for us during the early releases. He handled literally thousands of requests from people everywhere and saved the sanity of at least one of us. His calm good cheer was a foundation on which we could build.

Our thanks also goes to Todd Brunhoff (Tektronix) who was “loaned” to Project Athena at exactly the right moment to provide very capable and much-needed assistance during the alpha and beta releases. He was responsible for the successful integration of sources from multiple sites; we would not have had a release without him.

Our thanks also goes to Al Mento and Al Wojtas of Digital’s ULTRIX Documentation Group. With good humor and cheer, they took a rough draft and made it an infinitely better and more useful document. The work they have done will help many everywhere. We also would like to thank Hal Murray (Digital SRC) and Peter George (Digital VMS) who contributed much by proofreading the early drafts of this document.

Our thanks also goes to Jeff Dike (Digital UEG), Tom Benson, Jackie Granfield, and Vince Orgovan (Digital VMS) who helped with the library utilities implementation; to Hania Gajewska (Digital UEG-WSL) who, along with Ellis Cohen (CMU and Siemens), was instrumental in the semantic design of the window manager properties; and to Dave Rosenthal (Sun Microsystems) who also contributed to the protocol and provided the sample generic color frame buffer device-dependent code.

The alpha and beta test participants deserve special recognition and thanks as well. It is significant that the bug reports (and many fixes) during alpha and beta test came almost exclusively from just a few of the alpha testers, mostly hardware vendors working on product implementations of X. The continued public contribution of vendors and universities is certainly to the benefit of the entire X community.

Our special thanks must go to Sam Fuller, Vice-President of Corporate Research at Digital, who has remained committed to the widest public availability of X and who made it possible to greatly supplement MIT’s resources with the Digital staff in order to make version 11 a reality. Many of the people mentioned here are part of the Western Software Laboratory (Digital UEG-WSL) of the ULTRIX Engineering group and work for Smokey Wallace, who has been vital to the project’s success. Others not mentioned here worked on the toolkit and are acknowledged in the X Toolkit documentation.

Of course, we must particularly thank Paul Asente, formerly of Stanford University and now of Digital UEG-WSL, who wrote W, the predecessor to X, and Brian Reid, formerly of Stanford University and now of Digital WRL, who had much to do with W’s design.

Finally, our thanks goes to MIT, Digital Equipment Corporation, and IBM for providing the environment where it could happen.

Release 4

Our thanks go to Jim Fulton (MIT X Consortium) for designing and specifying the new Xlib functions for Inter-Client Communication Conventions (ICCCM) support.

We also thank Al Mento of Digital for his continued effort in maintaining this document and Jim Fulton and Donna Converse (MIT X Consortium) for their much-appreciated efforts in reviewing the changes.

Release 5

The principal authors of the Input Method facilities are Vania Joloboff (Open Software Foundation) and Bill McMahon (Hewlett-Packard). The principal author of the rest of the internationalization facilities is Glenn Widener (Tektronix). Our thanks to them for keeping their sense of humor through a long and sometimes difficult design process. Although the words and much of the design are due to them, many others have contributed substantially to the design and implementation. Tom McFarland (HP) and Frank Rojas (IBM) deserve particular recognition for their contributions. Other contributors were: Tim Anderson (Motorola), Alka Badshah (OSF), Gabe Beged-Dov (HP), Chih-Chung Ko (III), Vera Cheng (III), Michael Collins (Digital), Walt Daniels (IBM), Noritoshi Demizu (OMRON), Keisuke Fukui (Fujitsu), Hitoshi Fukumoto (Nihon Sun), Tim Greenwood (Digital), John Harvey (IBM), Hideki Hiura (Sun), Fred Horman (AT&T), Norikazu Kaiya (Fujitsu), Yuji Kamata (IBM), Yutaka Kataoka (Waseda University), Ranee Khubchandani (Sun), Akira Kon (NEC), Hiroshi Kuribayashi (OMRON), Teruhiko Kurosaka (Sun), Seiji Kuwari (OMRON), Sandra Martin (OSF), Narita Masahiko (Fujitsu), Masato Morisaki (NTT), Nelson Ng (Sun), Takashi Nishimura (NTT America), Makato Nishino (IBM), Akira Ohsone (Nihon Sun), Chris Peterson (MIT), Sam Shteingart (AT&T), Manish Sheth (AT&T), Muneiyoshi Suzuki (NTT), Cori Mehring (Digital), Shoji Sugiyama (IBM), and Eiji Tosa (IBM).

We are deeply indebted to Tatsuya Kato (NTT), Hiroshi Kuribayashi (OMRON), Seiji Kuwari (OMRON), Muneiyoshi Suzuki (NTT), and Li Yuhong (OMRON) for producing one of the first complete sample implementation of the internationalization facilities, and Hiromu Inukai (Nihon Sun), Takashi Fujiwara (Fujitsu), Hideki Hiura (Sun), Yasuhiro Kawai (Oki Technosystems Laboratory), Kazunori Nishihara (Fuji Xerox), Masaki Takeuchi (Sony), Katsuhisa Yano (Toshiba), Makoto Wakamatsu (Sony Corporation) for producing the another complete sample implementation of the internationalization facilities.

The principal authors (design and implementation) of the Xcms color management facilities are Al Tabayoyon (Tektronix) and Chuck Adams (Tektronix). Joann Taylor (Tektronix), Bob Toole (Tektronix), and Keith Packard (MIT X Consortium) also contributed significantly to the design. Others who contributed are: Harold Boll (Kodak), Ken Bronstein (HP), Nancy Cam (SGI), Donna Converse (MIT X Consortium), Elias Israel (ISC), Deron Johnson (Sun), Jim King (Adobe), Ricardo Motta (HP), Chuck Peek (IBM), Wil Plouffe (IBM), Dave Sternlicht (MIT X Consortium), Kumar Talluri (AT&T), and Richard Verberg (IBM).

We also once again thank Al Mento of Digital for his work in formatting and reformatting text for this manual, and for producing many pages. Thanks also to Clive Feather (IXI) for proof-reading and finding a number of small errors.

Release 6

Stephen Gildea (X Consortium) authored the threads support. Ovais Ashraf (Sun) and Greg Olsen (Sun) contributed substantially by testing the facilities and reporting bugs in a timely fashion.

The principal authors of the internationalization facilities, including Input and Output Methods, are Hideki Hiura (SunSoft) and Shigeru Yamada (Fujitsu OSS). Although the words and much

of the design are due to them, many others have contributed substantially to the design and implementation. They are: Takashi Fujiwara (Fujitsu), Yoshio Horiuchi (IBM), Makoto Inada (Digital), Hiromu Inukai (Nihon SunSoft), Song JaeKyung (KAIST), Franky Ling (Digital), Tom McFarland (HP), Hiroyuki Miyamoto (Digital), Masahiko Narita (Fujitsu), Frank Rojas (IBM), Hidetoshi Tajima (HP), Masaki Takeuchi (Sony), Makoto Wakamatsu (Sony), Masaki Wakao (IBM), Katsuhisa Yano (Toshiba) and Jinsoo Yoon (KAIST).

The principal producers of the sample implementation of the internationalization facilities are: Jeffrey Bloomfield (Fujitsu OSSI), Takashi Fujiwara (Fujitsu), Hideki Hiura (SunSoft), Yoshio Horiuchi (IBM), Makoto Inada (Digital), Hiromu Inukai (Nihon SunSoft), Song JaeKyung (KAIST), Riki Kawaguchi (Fujitsu), Franky Ling (Digital), Hiroyuki Miyamoto (Digital), Hidetoshi Tajima (HP), Toshimitsu Terazono (Fujitsu), Makoto Wakamatsu (Sony), Masaki Wakao (IBM), Shigeru Yamada (Fujitsu OSSI) and Katsuhisa Yano (Toshiba).

The coordinators of the integration, testing, and release of this implementation of the internationalization facilities are Nobuyuki Tanaka (Sony) and Makoto Wakamatsu (Sony).

Others who have contributed to the architectural design or testing of the sample implementation of the internationalization facilities are: Hector Chan (Digital), Michael Kung (IBM), Joseph Kwok (Digital), Hiroyuki Machida (Sony), Nelson Ng (SunSoft), Frank Rojas (IBM), Yoshiyuki Segawa (Fujitsu OSSI), Makiko Shimamura (Fujitsu), Shoji Sugiyama (IBM), Lining Sun (SGI), Masaki Takeuchi (Sony), Jinsoo Yoon (KAIST) and Akiyasu Zen (HP).

Jim Gettys
Cambridge Research Laboratory
Digital Equipment Corporation

Robert W. Scheifler
Laboratory for Computer Science
Massachusetts Institute of Technology

Chapter 1

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.

Xlib – C Language X Interface 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. *Xlib – C Language X Interface* 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 appropriate toolkit documentation. The *X Window System Protocol* provides the definitive word on the behavior 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
- Generic values and types
- Naming and argument conventions within Xlib
- Programming considerations
- Character sets and encodings
- Formatting conventions

1.1. Overview of the X Window System

Some of the terms used in this book are unique to X, and other terms that are common to other window systems have different meanings in X. You may find it helpful to refer to the glossary, which is located at the end of the book.

The X Window System supports one or more screens containing overlapping windows or subwindows. A screen is a physical monitor and hardware that can be color, grayscale, or monochrome. There can be multiple screens for each display or workstation. A single X server can provide display services for any number of screens. A set of screens for a single user with one keyboard and one pointer (usually a mouse) is called a display.

All the windows in an X server are arranged in strict hierarchies. At the top of each hierarchy is a root window, which covers each of the display screens. Each root window is partially or completely covered by child windows. All windows, except for root windows, have parents. There is usually at least one window for each application program. Child windows may in turn have their own children. In this way, an application program can create an arbitrarily deep tree on each screen. X provides graphics, text, and raster operations for 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 that 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 corner. 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 corner.

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.

1.2. 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 behavior (see section 11.8.1). 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.

1.3. 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 6 in this release of the standard. (Release 5 of Xlib was the first release to 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 6. 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, which are described in chapters 14 and 16. **<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, which are described in chapter 15. **<X11/Xlib.h>** must be included before including this file.
- **<X11/Xatom.h>**
This file declares all predefined atoms, which are symbols with the prefix “XA_”.
- **<X11/cursorfont.h>**
This file declares the cursor symbols for the standard cursor font, which are listed in appendix B. All cursor symbols have the prefix “XC_”.
- **<X11/keysymdef.h>**
This file declares all standard KeySym values, which are symbols with the 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_XKB_KEYS`, `XK_3270`, `XK_LATIN1`, `XK_LATIN2`, `XK_LATIN3`, `XK_LATIN4`, `XK_KATAKANA`, `XK_ARABIC`, `XK_CYRILLIC`, `XK_GREEK`, `XK_TECHNICAL`, `XK_SPECIAL`, `XK_PUBLISHING`, `XK_APL`, `XK_HEBREW`, `XK_THAI`, and `XK_KOREAN`.

- **<X11/keysym.h>**
This file defines the preprocessor symbols `XK_MISCELLANY`, `XK_XKB_KEYS`, `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, which are described in appendix C. 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, which are described in appendix D.

1.4. 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.

1.5. 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 lowercase for variables and all uppercase 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 lowercase. 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.

1.6. Programming Considerations

The major programming considerations are:

- Coordinates and sizes in X are actually 16-bit quantities. This decision was made 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 14 and the *Inter-Client Communication Conventions Manual*.

1.7. Character Sets and Encodings

Some of the Xlib functions make reference to specific character sets and character encodings. The following 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 set is the left/lower half of the graphic character set of ISO8859-1 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 the ISO8859-1 standard.
- **Latin Portable Character Encoding**
The encoding of the X Portable Character Set using the Latin-1 codepoints plus ASCII control characters. If a string is said to be in the Latin Portable Character Encoding, then it only contains characters from the X Portable Character Set, not all of Latin-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 ._-

1.8. Formatting Conventions

Xlib – C Language X Interface uses the following conventions:

- Global symbols are printed in **this special font**. These can be either function names, symbols defined in include files, or structure names. When declared and defined, function arguments are printed in *italics*. In the explanatory text that follows, they usually are printed in regular type.
- 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 11.8.2.
- 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.

Chapter 11

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.

11.1. 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.

11.2. 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 11.8.2). 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**.

11.3. 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 nonzero. 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 nonzero. 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 behavior 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.

11.4. 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

11.4.1. 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.

11.4.2. 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 if the event is useful without calling any Xlib functions. If the predicate directly or indirectly causes the state of the event queue to change, the result is not defined. If Xlib has been initialized for threads, the predicate is called with the display locked and the result of a call by the predicate to any Xlib function that locks the display is not defined unless the caller has first called **XLockDisplay**.

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.

11.4.3. 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.

11.5. 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**.

11.6. 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, or **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 10.3. 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; therefore these fields and the display field are ignored by **XSendEvent**.

XSendEvent returns zero if the conversion to wire protocol format failed and returns nonzero otherwise.

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

11.7. 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 traveled. 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, if the start time is later than the stop time, or if the start time is in the future, no events are returned; **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.

11.8. Handling Protocol Errors

Xlib provides functions that you can use to enable or disable synchronization and to use the default error handlers.

11.8.1. 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 behavior. 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 nonzero before starting a program under a debugger, will force synchronous library behavior.

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 procedure 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 behavior. If *onoff* is **False**, **XSynchronize** turns off synchronous behavior.

11.8.2. 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; the returned value is ignored. 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
------------	-------------

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.
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.

Error Code	Description
BadMatch	In a graphics request, the root and depth of the graphics context do not match those 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 text in the *default_string* argument is assumed to be in the encoding of the current locale, and the text stored in the *buffer_return* argument 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. In these names, uppercase and lowercase 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.