Programming with Xlib

Version 11, Release 3

HP 9000 Series 300/800 Computers

HP Part Number 98794-90002

HEWLETT
PACKARD

Hewlett-Packard Company
1000 NE Circle Blvd., Corvallis OR 97330

Certification for conformance with OSF/Motif user environment is pending.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

July 1988 . . . Edition 1
December 1988 . . . Edition 2
September 1989 . . . Edition 3

Contents

Introduction to Xlib 1

The X Window System is a network-transparent window system that was designed at MIT. It runs under 4.3BSD UNIX, ULTRIX-32, many other UNIX variants, VAX/VMS, MS/DOS, as well as several other operating systems.

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 manual 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 manual 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
- Naming and argument conventions
- Programming considerations
- Conventions used in this document

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, which can be either color or black and white. 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, 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. Each window has its own coordinate system. Child windows obscure their parents unless the child windows (of the same depth) have no background, and graphic operations in the parent window usually are clipped by the children.

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.

---

* The < > has the meaning defined by the # include statement of the C compiler and is a file relative to a well-known directory. On UNIX-based systems, this is */usr/include*.

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 8.12.1). 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 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.

1.4 Programming Considerations

The major programming considerations are:

- Keyboards are the greatest variable between different manufacturer's 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 9.
- Coordinates and sizes in X are actually 16-bit quantities. They usually are declared as an "int" in the interface (int is 16 bits on some machines). Values larger than 16 bits are truncated silently. Sizes (width and height) are unsigned quantities. This decision was taken to minimize the bandwidth required for a given level of performance.

1.5 Conventions Used in This Manual

This document uses the following conventions:

- Global symbols in this manual are printed in `this special font`. 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. 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 8.12.2.
- 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.
- Xlib defines the Boolean values of `True` and `False`.

Display Functions 2

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.

2.1 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 UNIX-based system, if the display_name is NULL, it defaults to the value of the DISPLAY environment variable.

On UNIX-based systems, the display name or DISPLAY environment variable is 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 2.2.1).

For example, the following would specify screen 2 of display 0 on the machine named mit-athena:

```
mit-athena:0.2
```

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 UNIX domain communications protocols. If the hostname is a host machine name and a single colon (:) separates the hostname and display number, `XOpenDisplay` connects using TCP streams, or UNIX domain IPC streams, if possible. If the environment variable XFORCE_INTERNET is set, TCP streams are used. If the hostname is local and a single colon (:) separates it from the display number, `XOpenDisplay` connects using UNIX domain IPC streams. If the hostname is not specified, Xlib uses whatever it believes is the fastest transport. 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 2.2.1.

X servers may implement various types of access control mechanisms (see section 7.11).

2.2 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 the:

- 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 not named in the conventional manner. Where these functions are mentioned, the terms should be interpreted as screen functions instead of display functions. For example, the *XDisplayWidth* function actually deals with screen width, not display width.

2.2.1 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;
```

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;
```

Both return the white pixel value for the specified screen.

```
ConnectionNumber(display)

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

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

```
DefaultColormap(display, screen_number)

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

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;
```

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`).

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

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

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;
```

Both return the root window for the default screen.

```
DefaultScreenOfDisplay(display)
```

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

Both return a pointer to the default screen.

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

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

Both return a pointer to the indicated screen.

```
DefaultScreen(display)
```

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

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;
```

Both return the default visual type for the specified screen. For further information about visual types, see section 3.1.

```
DisplayCells(display, screen_number)

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

Both return the number of entries in the default colormap.

```
DisplayPlanes(display, screen_number)

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

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;
```

Both return the string that was passed to `XOpenDisplay` when the current display was opened. On UNIX-based 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.

```
LastKnownRequestProcessed(display)

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

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;
```

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;
```

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

```
ProtocolRevision(display)

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

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

```
QLength(display)

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

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;
```

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;
```

Both return the number of available screens.

```
ServerVendor(display)

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

Both return a pointer to a null-terminated string that provides some identification of the owner of the X server implementation.

```
VendorRelease(display)

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

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

2.2.2 Image Format 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 sections 6.7 and 10.9).

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;
```

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;
```

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;
```

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;
```

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;
```

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;
```

Both return the height of the specified screen in millimeters.

```
DisplayWidth(display, screen_number)

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

Both return the width of the screen in pixels.

```
DisplayWidthMM(display, screen_number)

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

Both return the width of the specified screen in millimeters.

2.2.3 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;
```

Both return the black pixel value of the specified screen.

```
WhitePixelOfScreen(screen)
```

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

Both return the white pixel value of the specified screen.

```
CellsOfScreen(screen)
```

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

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

```
DefaultColormapOfScreen(screen)
```

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

Both return the default colormap of the specified screen.

```
DefaultDepthOfScreen(screen)
```

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

Both return the depth of the root window.

```
DefaultGCOfScreen(screen)
```

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

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;
```

Both return the default visual of the specified screen. For information on visual types, see section 3.1.

```
DoesBackingStore(screen)

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

Both return a value indicating whether the screen supports backing stores. The value returned can be one of `WhenMapped`, `NotUseful`, or `Always` (see section 3.2.4).

```
DoesSaveUnders(screen)

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

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 3.2.5).

```
DisplayOfScreen(screen)

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

Both return the display of the specified screen.

```
EventMaskOfScreen(screen)

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

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

```
WidthOfScreen(screen)

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

Both return the width of the specified screen in pixels.

```
HeightOfScreen(screen)

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

Both return the height of the specified screen in pixels.

```
WidthMMOfScreen(screen)

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

Both return the width of the specified screen in millimeters.

```
HeightMMOfScreen(screen)

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

Both return the height of the specified screen in millimeters.

```
MaxCmapsOfScreen(screen)

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

Both return the maximum number of installed colormaps supported by the specified screen (see section 7.3).

```
MinCmapsOfScreen(screen)

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

Both return the minimum number of installed colormaps supported by the specified screen (see section 7.3).

```
PlanesOfScreen(screen)

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

Both return the depth of the root window.

```
RootWindowOfScreen(screen)

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

Both return the root window of the specified screen.

2.3 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.

2.4 Freeing Client-Created Data

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

```
XFree(data)
     char *data;
```

data Specifies a pointer to 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.

2.5 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.

2.6 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 nonwindow 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 chapter 4).
- 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 3

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
- Translate window coordinates

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 9.1). 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.

3.1 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 2.2.1 and `XMatchVisualInfo`).

Xlib uses a `Visual` structure that contains information about the possible color mapping. The members of this structure pertinent to this discussion are class, red_mask, green_mask, blue_mask, bits_per_rgb, and map_entries. 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   |Static  |Pseudo |Static |Gray   |
  Colormap     |Color   |Color  |Gray   |Scale  |
               +-------------------------------+
Decomposed     |True    |Direct |
  Colormap     |Color   |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 map_entries 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.

3.2 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 4.2). 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 4.2).

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:

```
/* 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

3.2.1 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 chapter 8).

3.2.2 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.

3.2.3 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 chapter 8).

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.

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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

3.2.7 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 that window). 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. Both masks are the bitwise inclusive OR of one or more of the valid event mask bits. You can specify that no maskable events are reported by setting `NoEventMask` (default).

3.2.8 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 9).

3.2.9 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.

3.2.10 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.

3.3 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 predefined property functions (see chapter 9).

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.

`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 nonzero, 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 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 nonzero, 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.

3.4 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.

3.5 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 client's 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 3.2.8 and chapter 8. 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.

3.6 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.

3.7 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 nonzero, 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 nonzero, 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 3.2.3).

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 sta
`Opposite`	If the sibling occludes the window, the window is placed at the top of the stack. 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 sta
`Opposite`	If any sibling occludes the window, the window is placed at the top of the stack. I 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 a pointer to 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.

3.8 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.

3.9 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 3.2).

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

The current contents of the window are not changed by XSetWindowBackground or XSetWindowBackgroundPixmap

---

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.

3.10 Translating Window Coordinates

Applications, mostly window managers, often need to perform a coordinate transformation from the coordinate space of one window to another window or need to determine which subwindow a coordinate lies in. `XTranslateCoordinates` fulfills these needs (and avoids any race conditions) by asking the X server to perform this operation.

```
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.

The `XTranslateCoordinates` function 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 zero, 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.

Window Information Functions 4

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
- Manipulate property lists
- Obtain and change window properties
- Manipulate selections

4.1 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 a pointer to 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 nonzero if it succeeds. To free this list when it is no longer needed, use XFree.

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:

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

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 3.2.3.

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`.

To obtain the root window the pointer is currently on and the pointer coordinates relative to the root's origin, 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 7.4).

`XQueryPointer` can generate a `BadWindow` error.

4.2 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, see 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 4.3. For an explanation of the functions that let you get and set much of the information stored in these predefined properties, see chapter 9.

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	RGB_GREEN_MAP
CUT_BUFFER1	RGB_RED_MAP
CUT_BUFFER2	RESOURCE_MANAGER
CUT_BUFFER3	WM_CLASS
CUT_BUFFER4	WM_CLIENT_MACHINE
CUT_BUFFER5	WM_COMMAND
CUT_BUFFER6	WM_HINTS
CUT_BUFFER7	WM_ICON_NAME
RGB_BEST_MAP	WM_ICON_SIZE
RGB_BLUE_MAP	WM_NAME
RGB_DEFAULT_MAP	WM_NORMAL_HINTS
RGB_GRAY_MAP	WM_ZOOM_HINTS
	WM_TRANSIENT_FOR

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 6.5.

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. You should use a null-terminated ISO Latin-1 string for atom_name. 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. To free the resulting string, call XFree.

XGetAtomName can generate a BadAtom error.

4.3 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 4.2). 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 predefined property operations (see chapter 9).

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 a pointer to 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 a (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 2.5. 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 nonzero, 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.

4.4 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`.

In either event, the arguments are passed on unchanged. There are two predefined selection atoms: PRIMARY and SECONDARY.

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

Graphics Resource Functions 5

After you connect your program to the X server by calling `XOpenDisplay`, you can use the Xlib graphics resource functions to:

- Create, copy, and destroy colormaps
- Allocate, modify, and free color cells
- Read entries in a colormap
- Create and free pixmaps
- Create, copy, change, and destroy graphics contexts

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 6) take a GC as an argument. Although in theory it is possible to share 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.

Each X window always has an associated colormap that provides a level of indirection between pixel values and colors displayed on the screen. Many of the hardware displays built today have a single colormap, so the primitives are written to encourage sharing of colormap entries between applications. Because colormaps are associated with windows, X will support displays with multiple colormaps and, indeed, different types of colormaps. If there are not sufficient colormap resources in the display, some windows may not be displayed in their true colors. A client or window manager can control which windows are displayed in their true colors if more than one colormap is required for the color resources the applications are using.

Off-screen memory or pixmaps are often used to define frequently used images for later use in graphics operations. Pixmaps are also used to define tiles or patterns for use as window backgrounds, borders, or cursors. A single bit-plane pixmap is sometimes referred to as a bitmap.

Note that some screens have very limited off-screen memory. Therefore, you should regard off-screen memory as a precious resource.

Graphics operations can be performed to either windows or pixmaps, which collectively are called drawables. Each drawable exists on a single screen and can only be used on that screen. GCs can also only be used with drawables of matching screens and depths.

5.1 Colormap Functions

Xlib provides functions that you can use to manipulate a colormap. This section discusses how to:

- Create, copy, and destroy a colormap
- Allocate, modify, and free color cells
- Read entries in a colormap

The following functions 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. As each pixel is read out of display memory, its value is taken and looked up in the colormap. The values of the cell determine what color is displayed on the screen. On a multiplane display with a black-and-white monitor (with grayscale but not color), these values can be combined to determine the brightness on the screen.

Screens always have a default colormap, and programs typically allocate cells out of this colormap. You should not write applications that monopolize color resources. On a screen that either cannot load the colormap or cannot have a fully independent colormap, only certain kinds of allocations may work. Depending on the hardware, one or more colormaps may be resident (installed) at one time. To install a colormap, use `XInstallColormap`. The `DefaultColormap` macro returns the default colormap. The `DefaultVisual` macro returns the default visual type for the specified screen. Colormaps are local to a particular screen. Possible visual types are `StaticGray`, `GrayScale`, `StaticColor`, `PseudoColor`, `TrueColor`, or `DirectColor` (see section 3.1).

The functions discussed in this section operate on 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 scaled between 0 and 65535. Full color brightness is a value of 65535, independent of the number of bits actually used in the display hardware. Half brightness in a color is a value of 32767, and off is 0. This representation gives uniform results for color values across different screens. In some functions, the flags member controls which of the red, green, and blue members is used and can be one or more of `DoRed`, `DoGreen`, and `DoBlue`.

The members of the `Visual` structure that are pertinent to the discussion of `XCreateColormap` are class, red_mask, green_mask, blue_mask, and map_entries. The class member specifies the screen class and can be `GrayScale`, `PseudoColor`, `DirectColor`, `StaticColor`, `StaticGray`, or `TrueColor`. The red_mask, green_mask, and blue_mask members specify the color mask values. The map_entries member specifies the number of color map entries. The class member constant determines whether the initial values for map_entries are defined. If the class member is `GrayScale`, `PseudoColor`, or `DirectColor`, the initial values for map_entries are undefined. However, if the class member is `StaticColor`, `StaticGray`, or `TrueColor`, map_entries has initial values that are defined. However, these values are specific to the visual type and are not defined by the X server.

The class member constant also determines the constant you can pass to the alloc argument:

- If the class member is `StaticGray`, `StaticColor`, or `TrueColor`, you must pass `AllocNone`. Otherwise, a `BadMatch` error is generated.
- If the class member is any other class, you can pass `AllocNone`. In this case, the color map has no values defined for map_entries. This allows you and other clients to allocate the entries in the color map. You can also pass `AllocAll`. In this case, `XCreateColormap` allocates the entire color map as writable. The initial values of all map_entries are undefined. You cannot free any of these map_entries with a call to the function `XFreeColors`.

  When using `AllocAll` for a color map class of `GrayScale` or `PseudoColor`, the processing simulates a call to the function `XAllocColorCells`, where `XAllocColorCells` returns all pixel values from zero to N - 1. The value N represents the map_entries value in the specified `Visual` structure. For a color map class of `DirectColor`, the processing simulates a call to the function `XAllocColorPlanes`, where `XAllocColorPlanes` returns a pixel value of zero and rmask, gmask, and bmask values containing the same bits as the red_mask, green_mask, and blue_mask members in the specified `Visual` structure.

The introduction of color alters the view a programmer should take when dealing with a bitmap display. For example, when printing text, you write a pixel value, which is defined as a specific color, rather than setting or clearing bits. Hardware will impose limits (the number of significant bits, for example) on these values. Typically, one allocates color cells or sets of color cells. If read-only, the pixel values for these colors can be shared among multiple applications, and the RGB values of the cell cannot be changed. If read/write, they are exclusively owned by the program, and the color cell associated with the pixel value may be changed at will.

5.1.1 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 pointer to 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 3.1.

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 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 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.

5.1.2 Allocating, Modifying, and Freeing Color Cells

There are two ways of allocating color cells: explicitly as read-only entries by pixel value or read/write, where you can allocate a number of color cells and planes simultaneously. The read/write cells you allocate do not have defined colors until set with XStoreColor or XStoreColors.

To determine the color names, the X server uses a color database. Although you can change the values in a read/write color cell that is allocated by another application, this is considered "antisocial" behavior.

To allocate a read-only color cell, 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 values 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 values actually used. The corresponding colormap cell is read-only. In addition, XAllocColor returns nonzero if it succeeded or zero if it failed. Read-only colormap cells are shared among clients. 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 by name and return the closest color supported by the hardware, 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. You should use the ISO Latin-1 encoding; uppercase and lowercase do not matter.

XAllocNamedColor can generate a BadColor error.

To look up the name of a color, 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. You should use the ISO Latin-1 encoding; uppercase and lowercase do not matter. `XLookupColor` returns nonzero if the name existed in the color database or zero if it did not exist.

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 nonnegative, 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 nonzero 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 nonnegative.

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 nonnegative, 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 nonzero if it succeeded or zero if it failed.

`XAllocColorPlanes` can generate `BadColor` and `BadValue` errors.

To store RGB values into 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`, 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 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, or blue color components. You specify which color components are to be changed by setting DoRed, DoGreen, 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 set the color of a pixel to a named color, 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 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. You should use the ISO Latin-1 encoding; uppercase and lowercase do not matter.

XStoreNamedColor can generate BadAccess, BadColor, BadName, 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.

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), 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.

5.1.3 Reading Entries in a Colormap

The XQueryColor and XQueryColors functions return the RGB values stored in the specified colormap for the pixel value you pass in the pixel member of the XColor structure(s). 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 values of a single specified pixel value, 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 RGB values for each pixel in the XColor structures and sets the DoRed, DoGreen, and DoBlue flags.

XQueryColor can generate BadColor and BadValue errors.

To query the RGB values of an array of pixels stored in color structures, 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 values for each pixel in the XColor structures and sets the DoRed, DoGreen, and DoBlue flags.

XQueryColors can generate BadColor and BadValue errors.

5.2 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 nonzero, 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.

5.3 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	Hex Code	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 odd dashes meet.

`LineOnOffDash` — Only the even dashes are drawn, and cap-style applies to all internal ends 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 fina 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, center 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 fo a distance equal to half the line-width. (This is equivalent to `CapButt` fo 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 `Butt`/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-widt

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 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 nonzero, 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 one 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 one 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 one 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 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.

5.4 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

5.4.1 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.

5.4.2 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 nonzero, 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 each time a cap-style is applied at a line endpoint.

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 315 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.

5.4.3 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.

5.4.4 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.

Stipple depth is 1. The stipple and GC must be on the same screen, 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.

5.4.5 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.

5.4.6 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 `BadValue` 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 nondecreasing in their Y origin. YXSorted additionally constrains YSorted order in that all rectangles with an equal Y origin are nondecreasing 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 chapter 10.

5.4.7 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 6

Once you have connected the display to the X server, 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
- Manipulate cursors

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 number of requests sent to the server.

6.1 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.

6.2 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 scr, 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.

6.3 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 multiplied by 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.

6.3.1 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 a pointer to 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.

6.3.2 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 a pointer to 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 a pointer to 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.

6.3.3 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 a pointer to 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, 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.

6.3.4 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 multiplied by 64.

angle2 Specifies the path and extent of the arc relative to the start of the arc, in units of degrees multiplied by 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 a pointer to 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 $[x, y, width, height, angle1, angle2]$, 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 $[x, y, width, height, angle1, angle2]$, 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 $[0, 2\pi]$ 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 $[\frac{3\pi}{2}, 2\pi]$

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.

6.4 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

6.4.1 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 a pointer to 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.

6.4.2 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 a pointer to 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.
- If shape is Convex, the path is wholly convex. 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 behavior 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.

6.4.3 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 multiplied by 64.

angle2 Specifies the path and extent of the arc relative to the start of the arc, in units of degrees multiplied by 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 a pointer to 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.

6.5 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
- Set and retrieve the font search path
- 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 nonzero 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 nonzero, 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 nonzero 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 nondescending 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 nonkerned 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. When possible, fonts should have at least the properties listed in the following table. With atom names, uppercase and lowercase matter. The following built-in property atoms can be found in `<X11/Xatom.h>`:

Property Name	Type	Description
MIN_SPACE	unsigned	The minimum interword spacing, in pixels.
NORM_SPACE	unsigned	The normal interword spacing, in pixels.
MAX_SPACE	unsigned	The maximum interword spacing, in pixels.
END_SPACE	unsigned	The additional spacing at the end of sentences, in pixe
SUPERSCRIPT_X SUPERSCRIPT_Y	int	Offset from the character origin where superscripts sl begin, in pixels. If the origin is at [x,y], then superscri should begin at [x + SUPERSCRIPT_X, y - SUPERSCRIPT_Y].
SUBSCRIPT_X SUBSCRIPT_Y	int	Offset from the character origin where subscripts sho begin, in pixels. If the origin is at [x,y], then subscript should begin at [x + SUPERSCRIPT_X, y + SUPERSCRIPT_Y].
UNDERLINE_POSITION	int	Y offset from the baseline to the top of an underline, pixels. If the baseline is Y-coordinate y, then the top underline is at (y + UNDERLINE_POSITION).
UNDERLINE_THICKNESS	unsigned	Thickness of the underline, in pixels.
STRIKEOUT_ASCENT STRIKEOUT_DESCENT	int	Vertical extents for boxing or voiding characters, in pi If the baseline is at Y-coordinate y, then the top of th strikeout box is at (y - STRIKEOUT_ASCENT), and the height of the box is (STRIKEOUT_ASCENT + STRIKEOUT_DESCENT).
ITALIC_ANGLE	int	The angle of the dominant staffs of characters in the f degrees scaled by 64, relative to the three-o'clock pos from the character origin, with positive indicating counterclockwise motion (as in `XDrawArc`).
X_HEIGHT	int	1 ex as in TeX, but expressed in units of pixels. Often height of lowercase x.
QUAD_WIDTH	int	1 em as in TeX, but expressed in units of pixels. Ofte width of the digits 0-9.
CAP_HEIGHT	int	Y offset from the baseline to the top of the capital let ignoring accents, in pixels. If the baseline is at Y-coo y, then the top of the capitals is at

		(y - CAP_HEIGHT).
WEIGHT	unsigned	The weight or boldness of the font, expressed as a valı between 0 and 1000.
POINT_SIZE	unsigned	The point size of this font at the ideal resolution, expr in 1/10 points.
RESOLUTION	unsigned	The number of pixels per point, expressed in 1/100, at which this font was created.

6.5.1 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. The name should be ISO Latin-1 encoding; uppercase and lowercase do not matter. 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). 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 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.

6.5.2 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 string should be ISO Latin-1; uppercase and lowercase do not matter. Each string is terminated by an ASCII null. 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. 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 a pointer to 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. To free the allocated name array, the client should call XFreeFontNames. To free the 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 pointer to the font information returned by XListFontsWithInfo.

actual_count Specifies the actual number of matched font names returned by XListFontsWithInfo.

6.5.3 Setting and Retrieving the Font Search Path

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 interpretation of the strings is operating system dependent, but they are intended to specify directories to be searched in the order listed. Also, the contents of these strings are operating system dependent and are not intended to be used by client applications. Usually, the X server is free to cache font information internally rather than having to read fonts from files. In addition, 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 operating system 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. 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.

6.5.4 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.

6.5.5 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 a pointer to 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 a pointer to 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.

6.5.6 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.

XQueryTextExtents and XQueryTextExtents16 can generate BadFont and BadGC errors.

6.6 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.

6.6.1 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 a pointer to 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 a pointer to 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.

6.6.2 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.

6.6.3 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.

6.7 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 chapter 10).

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;
      char *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 height, width, or xoffset be changed when the image is sent to the server. That is, you may send a subset of the image. All other members are characteristics of both the image and the server, and should not be changed. 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 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 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;
        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 `XYBitmap`, `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 noninferior 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.

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 XYBitmap, 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 noninferior 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.

XGetSubImage can generate BadDrawable, BadGC, BadMatch, and BadValue errors.

6.8 Cursors

This section discusses how to:

- Create a cursor
- Change or destroy a cursor
- Define the cursor for a window

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. It is intended that most standard cursors will be stored as a special font.

6.8.1 Creating a Cursor

Xlib provides functions that you can use to create a font, bitmap, or glyph cursor.

To create a cursor from a standard 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 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 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 coincidentally 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.

6.8.2 Changing and Destroying Cursors

Xlib provides functions that you can use to change the cursor color, destroy the cursor, and determine the best cursor size.

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.

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.

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.

6.8.3 Defining the Cursor

Xlib provides functions that you can use to define or undefine the cursor that should be displayed in a window.

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` 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 Manager Functions 7

Although it is difficult to categorize functions as application only or window manager only, the functions in this chapter are most often used by window managers. It is not expected that these functions will be used by most application programs. You can use the Xlib window manager functions to:

- Change the parent of a window
- Control the lifetime of a window
- Determine resident colormaps
- Grab the pointer
- Grab the keyboard
- Grab the server
- Control event processing
- Manipulate the keyboard and pointer settings
- Control the screen saver
- Control host access

7.1 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 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.

7.2 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 2.6. 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.

7.3 Determining Resident Colormaps

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.

7.4 Pointer Grabbing

Xlib provides functions that you can use to control input from the pointer, which usually is a mouse. Window managers most often use these facilities to implement certain styles of user interfaces. Some toolkits also need to use these facilities for special purposes.

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 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 behavior. 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.

7.5 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 pointer events are to be reported as usual or reported with respect to the grab window if selected by the event mask.

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 pointer events are to be reported as usual or reported with respect to the grab window if selected by the event mask.

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.

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 usual. If the pointer is frozen twice by the client on behalf of two sepa grabs, AsyncPointer thaws for both. AsyncPointer has no eff the pointer is not frozen by the client, but the pointer need not be grab 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 poi again appears to freeze. However, if the reported event causes the poi grab to be released, the pointer does not freeze. SyncPointer has effect if the pointer is not frozen by the client or if the pointer is not gr by the client.

ReplayPointer If the pointer is actively grabbed by the client and is frozen as the resul 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 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 of an event.

`AsyncKeyboard` — If the keyboard is frozen by the client, keyboard event processing continu as usual. If the keyboard is frozen twice by the client on behalf of two separate grabs, `AsyncKeyboard` thaws for both. `AsyncKeyboard` h 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 eve processing continues as usual until the next `KeyPress` or `KeyReleas` event is reported to the client. At this time, the keyboard again appears t freeze. However, if the reported event causes the keyboard grab to be released, the keyboard does not freeze. `SyncKeyboard` has no effect i the keyboard is not frozen by the client or if the keyboard is not grabbed b the client.

`ReplayKeyboard` — If the keyboard is actively grabbed by the client and is frozen as the result 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 released and that event is completely reprocessed. This time, however, th 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 th result of an event.

`SyncBoth` — If both pointer and keyboard are frozen by the client, event processing fo both devices continues as usual until the next `ButtonPress`, `ButtonRelease`, `KeyPress`, or `KeyRelease` event is reported to th 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 t reported event causes the grab to be released, then the devices do not fre (but if the other device is still grabbed, then a subsequent event for it will cause both devices to freeze). `SyncBoth` has no effect unless both point and keyboard are frozen by the client. If the pointer or keyboard is froze twice by the client on behalf of two separate grabs, `SyncBoth` thaws for both (but a subsequent freeze for `SyncBoth` will only freeze each devic once).

`AsyncBoth` — If the pointer and the keyboard are frozen by the client, event processing both devices continues as usual. If a device is frozen twice by the client o behalf of two separate grabs, `AsyncBoth` thaws for both. `AsyncBot` 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.

XAllowEvents can generate a BadValue error.

7.6 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.

7.7 Miscellaneous Control Functions

This section discusses how to:

- Control the input focus
- Control the pointer
- Kill clients

7.7.1 Controlling Input Focus

Xlib provides functions that you can use to move the pointer position as well as to set and get the input focus.

To move the pointer to an arbitrary point on the screen, 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 specified rectangle 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.

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.

7.7.2 Killing Clients

Xlib provides functions that you can use to control the lifetime of resources owned by a client or to cause the connection to a client to be destroyed.

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 2.6.

XSetCloseDownMode can generate a BadValue error.

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 2.6). 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.

7.8 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 behavior, and so on. The default values for many of these functions are determined by command line arguments to the X server and, on UNIX-based 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 one value for each bit set to 1 in the mask.

values Specifies which controls to change. This mask is the bitwise inclusive OR of the valid control mask bits.

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 nonnegative 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`. Elements of the list are indexed starting from one. The length of the list must be the same as `XGetPointerMapping` would return, or a `BadValue` error results. The index is a core button number, and the element of the list defines the effective number. 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 nonzero 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. The list contains the mapping, starting with button 1. `XGetPointerMapping` returns the number of physical buttons actually on the pointer. The nominal mapping for a pointer is the identity mapping, where button [i] has the value i. 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.

7.9 Keyboard Encoding

Most applications will find the simple interface `XLookupString`, which performs simple translation of a key event to an ASCII string, most useful. Keyboard-related utilities are discussed in chapter 10. The following section explains how to completely control the bindings of symbols to keys and modifiers.

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 include 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 length of the list can vary with each KeyCode. The list is intended to convey the set of symbols on the corresponding key. By convention, if the list contains a single KeySym and if that KeySym is alphabetic and case distinction is relevant for it, then it should be treated as equivalent to a two-element list of the lowercase and uppercase KeySyms. For example, if the list contains the single KeySym for uppercase *A*, the client should treat it as if it were a pair with lowercase *a* as the first KeySym and uppercase *A* as the second KeySym.

For any KeyCode, the first KeySym in the list should be chosen as the interpretation of a KeyPress when no modifier keys are down. The second KeySym in the list normally should be chosen when the Shift modifier is on or when the Lock modifier is on and Lock is interpreted as ShiftLock. When the Lock modifier is on and is interpreted as CapsLock, it is suggested that the Shift modifier first be applied to choose a KeySym. However, if that KeySym is lowercase alphabetic, the corresponding uppercase KeySym should be used instead. Other interpretations of CapsLock are possible; for example, it may be viewed as equivalent to ShiftLock, but only applying when the first KeySym is lowercase alphabetic and the second KeySym is the corresponding uppercase alphabetic. No interpretation of KeySyms beyond the first two in a list is suggested here. 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.

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 a pointer to 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 a pointer to 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 a pointer to 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 a pointer to 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 a pointer to 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 nonzero KeyCodes have meaning in each set, and zero KeyCodes are ignored. In addition, all of the nonzero KeyCodes must be in the range specified by min_keycode and max_keycode in the

Display structure, or a BadValue error results. No KeyCode may appear twice in the entire map, 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.

7.10 Screen Saver Control

Xlib provides functions that you can use to set, force, activate, or reset the screen saver and to obtain the current screen saver values.

To set the screen saver, 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 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, 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 minutes, 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`).

7.11 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 protection, however, connections are permitted only from machines you trust. This is adequate on single-user workstations but 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 UNIX-based systems, each host is listed in .PN /etc/X?.hosts; ? indicates the display number. This file consists of host names separated by newlines. DECnet nodes must terminate in :: to tell 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.

Servers also can implement other access control policies in addition to or in place of this host access facility. See "X Window System Protocol" for further information.

7.11.1 Adding, Getting, or Removing Hosts

Xlib has functions for adding, getting, or removing hosts from the access control list. 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 named host to the access control list for that display. A `BadAccess` error results if the server and the client issuing the command are not the same host.

`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 connect. It also returns a pointer to a list of host structures allocated by the function. Free this memory when not needed 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 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 operates under the same constraints as the `XRemoveHosts` function, and can generate the same errors.

7.11.2 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.

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 and Event-Handling Functions 8

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 begins with a discussion of the following topics associated with events:

- Event types
- Event structures
- Event mask
- Event processing

It then 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 error events

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.

8.1 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 8.4.

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

8.2 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 8.7).

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.

8.3 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 4.4). 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

8.4 Event Processing

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	Expose	XExposeEvent	

GCGraphicsExposures in GC	GraphicsExpose	XGraphicsExposeEvent	
	NoExpose	XNoExposeEvent	
FocusChangeMask	FocusIn	XFocusInEvent	XFocusChangeEvent
	FocusOut	XFocusOutEvent	XFocusChangeEvent
KeymapStateMask	KeymapNotify	XKeymapEvent	
KeyPressMask	KeyPress	XKeyPressedEvent	XKeyEvent
KeyReleaseMask	KeyRelease	XKeyReleasedEvent	XKeyEvent
OwnerGrabButtonMask	N.A.	N.A.	
PointerMotionMask	MotionNotify	XPointerMovedEvent	XMotionEvent
PointerMotionHintMask	N.A.	N.A.	
PropertyChangeMask	PropertyNotify	XPropertyEvent	
ResizeRedirectMask	ResizeRequest	XResizeRequestEvent	
StructureNotifyMask	CirculateNotify	XCirculateEvent	
	ConfigureNotify	XConfigureEvent	
	DestroyNotify	XDestroyWindowEvent	
	GravityNotify	XGravityEvent	
	MapNotify	XMapEvent	
	ReparentNotify	XReparentEvent	
	UnmapNotify	XUnmapEvent	
SubstructureNotifyMask	CirculateNotify	XCirculateEvent	
	ConfigureNotify	XConfigureEvent	
	CreateNotify	XCreateWindowEvent	
	DestroyNotify	XDestroyWindowEvent	
	GravityNotify	XGravityEvent	
	MapNotify	XMapEvent	
	ReparentNotify	XReparentEvent	
	UnmapNotify	XUnmapEvent	
SubstructureRedirectMask	CirculateRequest	XCirculateRequestEvent	
	ConfigureRequest	XConfigureRequestEvent	
	MapRequest	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

8.4.1 Keyboard and Pointer Events

This section discusses:

- Pointer button events
- Keyboard and pointer events

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.

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 10.

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.

- `PointerMotionHint`

  If `PointerMotionHintMask` is selected, 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 member or an ancestor of it. 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 chapter 7).

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`.

8.4.2 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`.

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.

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 8.4.2.1) 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 8.4.2.1) 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.

8.4.3 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`.

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.

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.

8.4.4 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.

8.4.5 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.

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) honoring 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 nonzero, 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 nonzero, 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 nonzero counts and perform full redisplays on events with zero counts.

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 nonzero, 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 nonzero, at least that number of GraphicsExpose events (and possibly more) are to follow for this window.

8.4.6 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

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.

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, 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, 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.

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 nonzero. 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.

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.

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.

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.

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.

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`.

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.

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`.

8.4.7 Structure Control Events

This section discusses:

- `CirculateRequest` events
- `ConfigureRequest` events
- `MapRequest` events
- `ResizeRequest` events

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. To initiate a circulate window request on the window, the client calls `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, 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.

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, 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.

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, assume 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.

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.

8.4.8 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.

8.4.9 Client Communication Events

This section discusses:

- ClientMessage events
- PropertyNotify events
- SelectionClear events
- SelectionNotify events
- SelectionRequest events

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 window member is set to the window to which the event was sent. 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 message_type or data members.

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 PropertyDeleted */
} 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 PropertyDeleted when a property of the window is deleted using XDeleteProperty or, if the delete argument is True, XGetWindowProperty.

SelectionClear Events

The X server reports SelectionClear events to the current owner of a selection. The X server generates this event type on the window losing ownership of the selection to a new owner. This sequence of events could occur whenever a client calls 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 window member is set to the window losing ownership of the selection. The selection member is set to the selection atom. The time member is set to the last change time recorded for the selection. The owner member is the window that was specified by the current owner in its XSetSelectionOwner call.

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 and the specified selection is owned by a window.

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 owning the selection and is 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 time and is a timestamp or `CurrentTime` from the `ConvertSelection` request.

The client who owns the selection should do the following:

- The owner client should convert the selection based on the atom contained in the target member.
- If a property was specified (that is, the property member is set), the owner client should store the result as that property on the requestor window and then send a `SelectionNotify` event to the requestor by calling `XSendEvent` with an empty event-mask; that is, the event should be sent to the creator of the requestor window.
- If `None` is specified as the property, the owner client should choose a property name on the requestor window and then send a `SelectionNotify` event giving the actual name.
- If the selection cannot be converted as requested, the owner client should send a `SelectionNotify` event with the property set to `None`.

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`.

8.5 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.

8.6 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 error event received by Xlib, XSync calls the client application's error handling routine (see section 8.12.2). 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.

8.7 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.

8.8 Manipulating the Event Queue

Xlib provides functions that let you manipulate the event queue. The next three sections discuss 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

8.8.1 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.

8.8.2 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;
     char *arg;
```

display Specifies the connection to the X server.

event Specifies a pointer to 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)();
      char *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)();
      char *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)();
      char *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.

8.8.3 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.

8.9 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 a pointer to 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.

8.10 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 a pointer to 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 8.3. 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 nonzero otherwise.

XSendEvent can generate BadValue and BadWindow errors.

8.11 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 size of 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 start time is later than the stop time or if the start time is in the future, no events are returned. 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.

8.12 Handling Error Events

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

8.12.1 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 UNIX-based 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 function to be called after an Xlib function that generates a protocol request completes its work.

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.

8.12.2 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 error events 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`.

```
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 `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 grabb by another client. A client attempts to free a colormap entry that it had not already allocated. A client attempts to store into a read-only or unallocated colorma 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 o allocation space in the middle of service. The semantics when a server runs out of allocation space are left unspecified, but a serve may generate a BadAlloc error on any request for this reason, and clients should be prepared to receive such errors and handle o 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 serve 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. 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 message (or the default message) from the error message database. Xlib uses this function internally to look up its error messages. On a UNIX-based system, the error message database is `/usr/lib/X11/XErrorDB`.

The name argument should generally be the name of your application. The message argument should indicate which type of error message you want. Xlib uses three predefined message types to report errors (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 The major 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`.

```
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.

Predefined Property Functions 9

There are a number of predefined properties for information commonly associated with windows. The atoms for these predefined properties can be found in `<X11/Xatom.h>`, where the prefix XA_ is added to each atom name.

Xlib provides functions that you can use to perform operations on predefined properties. This chapter discusses how to:

- Communicate with window managers
- Manipulate standard colormaps

9.1 Communicating with Window Managers

This section discusses a set of properties and functions that are necessary for clients to communicate effectively with window managers. Some of these properties have complex structures. Because all the data in a single property on the server has to be of the same format (8-bit, 16-bit, or 32-bit) and because the C structures representing property types cannot be guaranteed to be uniform in the same way, Set and Get functions are provided for properties with complex structures.

These functions define but do not enforce minimal policy among window managers. Writers of window managers are urged to use the information in these properties rather than invent their own properties and types. A window manager writer, however, can define additional properties beyond this least common denominator.

In addition to Set and Get functions for individual properties, Xlib includes one function, `XSetStandardProperties`, that sets all or portions of several properties. Applications are encouraged to provide the window manager more information than is possible with `XSetStandardProperties`. To do so, they should call the Set functions for the additional or specific properties that they need.

Every application should specify the following information:

- Name of the application
- Name to be used in the icon

- Command used to invoke the application
- Size and window manager hints

Xlib does not set defaults for the properties described in this section. Thus, the default behavior is determined by the window manager and may be based on the presence or absence of certain properties. All the properties are considered to be hints to a window manager. When implementing window management policy, a window manager determines what to do with this information and can ignore it.

The supplied properties are:

Name	Type	Format	Description
WM_NAME	STRING	8	Name of the application.
WM_ICON_NAME	STRING	8	Name to be used in icon.
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_ZOOM_HINTS	WM_SIZE_HINTS	32	Size hints for a zoomed window. The C type of this property is `XSizeHints`.
WM_HINTS	WM_HINTS	32	Additional hints set by client for use by the window manager. The C type of this property is `XWMHints`.
WM_COMMAND	STRING	8	The command and arguments, separated by ASCII nulls, used to invoke the application.
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`.
WM_CLASS	STRING	32	Set by application programs to allow window and session managers to obtain the application's resources from the resource database.
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, is not really a normal application window.

The atom names stored in <`X11/Xatom.h`> are named XA_*PROPERTY_NAME*.

Xlib provides functions that you can use to set and get predefined properties. Note that calling the Set function for a property with complex structure redefines all members in that property, even though only some of those members may have a specified new value. Simple properties for which Xlib does not provide a Set or Get function can be set by using `XChangeProperty`, and their values can be retrieved using `XGetWindowProperty`. The remainder of this section discusses how to:

- Set standard properties
- Set and get the name of a window
- Set and get the icon name of a window
- Set the command and arguments of the application
- Set and get window manager hints
- Set and get window size hints
- Set and get icon size hints
- Set and get the class of a window
- Set and get the transient property for a window

9.1.1 Setting Standard Properties

Use `XSetStandardProperties` to specify a minimum set of properties describing the "quickie" application. This function 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 (null-terminated string).
icon_name	Specifies the icon name (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. (Typically, the main program argv array.)

argc Specifies the number of arguments.

hints Specifies a pointer to the size hints for the window in its normal state.

Use XSetStandardProperties to allow simple applications to set the most essential properties with a single call. Use XSetStandardProperties to give a window manager some information about your program's preferences. However, don't use this function with applications that need to communicate more information than the function can handle.

XSetStandardProperties can generate BadAlloc and BadWindow errors.

9.1.2 Setting and Getting Window Names

Xlib provides functions that you can use to set and read the name of a window. These functions set and read the WM_NAME property.

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 window name (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.

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 pointer to window name (null-terminated string).

The `XFetchName` function returns the name of the specified window. If it succeeds, it returns nonzero; if no name is set for the window, it returns zero. If the WM_NAME property has not been set for this window, `XFetchName` sets window_name_return to NULL. When finished with it, a client uses `XFree` to release the window name string.

`XFetchName` can generate a `BadWindow` error.

9.1.3 Setting and Getting Icon Names

Xlib provides functions that you can use to set and read the name to be displayed in a window's icon. These functions set and read the WM_ICON_NAME property.

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 icon name (null-terminated string).

`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 pointer to window's icon name (null-terminated string).

The `XGetIconName` function returns the name for display in the specified window's icon. If it succeeds, it returns nonzero; if no icon name is set for the window, it returns zero. If no name is assigned to the window, `XGetIconName` sets icon_name_return to NULL. A client uses `XFree` to release the icon name string.

XGetIconName can generate a BadWindow error.

9.1.4 Setting the Command

To set the command property, use XSetCommand. This function sets the WM_COMMAND property.

```
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.

XSetCommand sets the command and arguments used to invoke the application.

XSetCommand can generate BadAlloc and BadWindow errors.

9.1.5 Setting and Getting Window Manager Hints

The functions discussed in this section set and read the WM_HINTS property and use the flags and the XWMHints structure, as defined in the <X11/Xutil.h> header file:

```
/* 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 X10-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   DontCareState   0   /* don't know or care */
#define   NormalState     1   /* most applications start this way */
#define   ZoomState       2   /* application wants to start zoomed */
#define   IconicState     3   /* application wants to start as an icon */
#define   InactiveState   4   /* application believes it is seldom used;
                                 some wm's may put it on inactive menu */
```

The icon_mask specifies which pixels of the icon_pixmap should be used as the icon. This allows for nonrectangular icons. Both the 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 the window manager hints for a window, 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 a pointer to the window manager hints.

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 the window manager hints for a window, 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 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.

9.1.6 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:

```
/* 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   PAllHints                (PPosition|PSize|PMinSize|PMaxSize|
                                   PResizeInc|PAspect)
```

```
/* Values */

typedef struct {
      long flags;            /* marks which fields in this structure are defined */
      int x, y;
      int width, height;
      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;
} XSizeHints;
```

The x, y, width, and height members describe a desired position and size for the window. To indicate that this information was specified by the user, set the `USPosition` and `USSize` flags. To indicate that it was specified by the application without any user involvement, set `PPosition` and `PSize`. This lets a window manager know that the user specifically asked where the window should be placed or how the window should be sized and that the window manager does not have to rely on the program's opinion.

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 next two functions set and read the WM_NORMAL_HINTS property.

To set the size hints for a given window in its normal state, use `XSetNormalHints`.

```
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`.

```
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 nonzero 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`.

```
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`.

```
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 nonzero 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`.

```
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.

```
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 nonzero status if a size hint was defined or zero otherwise.

XGetSizeHints can generate BadAtom and BadWindow errors.

9.1.7 Setting and Getting Icon Size Hints

Applications can cooperate with window managers by providing icons in sizes supported by a window manager. To communicate the supported icon sizes to the applications, a window manager should set the icon size property on the root window of the screen. To find out what icon sizes a window manager supports, applications should read the icon size property from the root window of the screen.

The functions discussed in this section set or read the WM_ICON_SIZE property. In addition, they use the XIconSize structure, which is defined in <X11/Xutil.h> and 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 the icon size hints for a window, 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 a pointer to 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 return the icon sizes hints for a window, 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 a pointer to 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 or nonzero otherwise. `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.

9.1.8 Setting and Getting the Class of a Window

Xlib provides functions to set and get the class of a window. These functions set and read the WM_CLASS property. In addition, they use the `XClassHint` structure, which is defined in `<X11/Xutil.h>` and 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 the class of a window, 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 a pointer to a `XClassHint` structure that is to be used.

The `XSetClassHint` function sets the class hint for the specified window.

`XSetClassHint` can generate `BadAlloc` and `BadWindow` errors.

To get the class of a window, 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 of the specified window. To free res_name and res_class when finished with the strings, use `XFree`.

`XGetClassHint` can generate a `BadWindow` error.

9.1.9 Setting and Getting the Transient Property

An application may want to indicate to the window manager that a transient, top-level window (for example, a dialog box) is operating on behalf of (or is transient for) another window. To do so, the application would set the WM_TRANSIENT_FOR property of the dialog box to be the window ID of its main window. Some window managers use this information to unmap an application's dialog boxes (for example, when the main application window gets iconified).

The functions discussed in this section set and read the WM_TRANSIENT_FOR property.

To set the WM_TRANSIENT_FOR property for a window, 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 get the WM_TRANSIENT_FOR value for a window, 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.

`XGetTransientForHint` can generate a `BadWindow` error.

9.2 Manipulating 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 3D 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 (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. He invokes a color palette program to mix and choose colors. The color palette program also needs a colormap with 8 reds, 8 greens, and 4 blues, so just as 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.

As another example, 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.

9.2.1 Standard Colormaps

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. If the standard colormaps do not exist, you should create them by opening a new connection, creating the properties, and setting the close-down mode of the connection to `RetainPermanent`.

The `XStandardColormap` structure contains:

```
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;
} 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
```

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
```

The properties containing the `XStandardColormap` information have the type RGB_COLOR_MAP.

9.2.2 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 `XStandardColormap`.

The property defines an RGB subset of the default colormap of the screen. 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 3D 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.

On an 8-plane `PseudoColor` display, RGB_BEST_MAP should be a 3/3/2 allocation. On a 24-plane `DirectColor` display, RGB_BEST_MAP should be an 8/8/8 allocation. On other displays, the RGB_BEST_MAP allocation is purely up to the implementor of the display.

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.

9.2.3 Getting and Setting an XStandardColormap Structure

To get the XStandardColormap structure associated with one of the described atoms, use XGetStandardColormap.

```
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. 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);
```

Once you have fetched a standard colormap, you can use it to convert RGB values into pixel values. For example, given an XStandardColormap structure and floating-point RGB coefficients in the range 0.0 to 1.0, you can compose pixel values with the following C expression:

```
pixel = base_pixel
      + ((unsigned long) (0.5 + r * red_max)) * red_mult
      + ((unsigned long) (0.5 + g * green_max)) * green_mult
      + ((unsigned long) (0.5 + b * blue_max)) * blue_mult;
```

The use of addition rather than logical OR for composing pixel values permits allocations where the RGB value is not aligned to bit boundaries.

XGetStandardColormap can generate BadAtom and BadWindow errors.

To set a standard colormap, use XSetStandardColormap.

```
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 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 (not required for RGB_DEFAULT_MAP)
   - Determine the color capabilities of the display.
   - 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.

`XSetStandardColormap` can generate `BadAlloc`, `BadAtom`, and `BadWindow` errors.

Application Utility Functions 10

Once you have initialized the X system, you can use the Xlib utility functions to:

- Handle keyboard events
- Obtain the X environment defaults
- Parse window geometry strings
- Parse hardware colors strings
- Generate regions
- Manipulate regions
- Use cut and paste buffers
- Determine the appropriate visual
- Manipulate images
- Manipulate bitmaps
- Use the resource manager
- 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.

10.1 Keyboard Utility Functions

This section discusses keyboard event functions and KeySym classification macros.

10.1.1 Keyboard Event Functions

The X server does not predefine the keyboard to be ASCII characters. It is often useful to know that the *a* key was just pressed or that it was just released. When a key is pressed or released, the X server sends keyboard events to client programs. The structures associated with keyboard events contain a keycode member that assigns a number to each physical key on the keyboard. For a discussion of keyboard event processing, see section 8.4.1. For information on how to manipulate the keyboard encoding, see section 7.9.

Because KeyCodes are completely arbitrary and may differ from server to server, client programs wanting to deal with ASCII text, for example, must explicitly convert the KeyCode value into ASCII. Therefore, Xlib provides functions to help you customize the keyboard layout. Keyboards differ dramatically, so writing code that presumes the existence of a particular key on the main keyboard creates portability problems.

Keyboard events are usually sent to the deepest viewable window underneath the pointer's position that is interested in that type of event. It is also possible to assign the keyboard input focus to a specific window. When the input focus is attached to a window, keyboard events go to the client that has selected input on that window rather than the window under the pointer.

The functions in this section handle the shift modifier computations suggested by the protocol. The KeySym table is internally modified to define the lowercase transformation of a-z by adding the lowercase KeySym to the first element of the KeySym list (used internally) defined for the KeyCode, when the list is of length 1. If you want the untransformed KeySyms defined for a key, you should only use the functions described in section 7.9.

To look up the KeySyms, 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 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.

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 is a convenience routine that maps a key event to an ISO Latin-1 string, using the modifier bits in the key event to deal with shift, lock, and control. It returns the translated string into the user's buffer. It also detects any rebound KeySyms (see XRebindKeysym) and returns the specified bytes. XLookupString returns the length of the string stored in the tag buffer. If the lock modifier has the caps lock KeySym associated with it, XLookupString interprets the lock modifier to perform caps lock processing.

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.

To rebind the meaning of a KeySym for a client, use XRebindKeysym.

```
XRebindKeysym(display, keysym, list, mod_count, string, bytes_string)
      Display *display;
      KeySym keysym;
      KeySym list[];
      int mod_count;
      unsigned char *string;
      int bytes_string;
```

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 a pointer to the string that is copied and will be returned by `XLookupString`.
bytes_string	Specifies the length of the string.

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. Note that you can rebind a KeySym that may not exist.

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.

Valid KeySym names are listed in `<X11/keysymdef.h>` by removing the XK_ prefix from each name. 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. If the specified KeySym is not defined, `XKeysymToString` returns a NULL.

To convert a key code to a defined KeySym, 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 convert a KeySym to the appropriate KeyCode, 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.

10.1.2 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)
```

Returns True if the specified KeySym is a cursor key.

```
IsFunctionKey(keysym)
```

Returns True if the specified KeySym is a function key.

```
IsKeypadKey(keysym)
```

Returns True if the specified KeySym is a keypad key.

```
IsMiscFunctionKey(keysym)
```

Returns True if the specified KeySym is a miscellaneous function key.

`IsModifierKey(`*keysym*`)`

Returns `True` if the specified KeySym is a modifier key.

`IsPFKey(`*keysym*`)`

Returns `True` if the specified KeySym is a PF key.

10.2 Obtaining the X Environment Defaults

A program often needs a variety of options in the X environment (for example, fonts, colors, mouse, background, text, and cursor). Specifying these options on the command line is inefficient and unmanageable because individual users have a variety of tastes with regard to window appearance. `XGetDefault` makes it easy to find out the fonts, colors, and other environment defaults favored by a particular user. Defaults are usually loaded into the RESOURCE_MANAGER property on the root window at login. If no such property exists, a resource file in the user's home directory is loaded. On a UNIX-based 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. For details of the format of these files, see section 10.11.

The `XGetDefault` function provides a simple interface for clients not wishing to use the X toolkit or the more elaborate interfaces provided by the resource manager discussed in section 10.11.

```
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 NULL if the option name specified in this argument does not exist for the program. The strings returned by `XGetDefault` are owned by Xlib and should not be modified or freed by the client.

To obtain a pointer to the resource manager string 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`.

10.3 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;
     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>x<height>][{+-}<xoffset>{+-}<yoffset>]
```

The items in this form 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.)

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 parse window geometry given a user-specified position and a default position, use XGeometry.

```
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.

10.4 Parsing the Color Specifications

To parse color values, 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 provides a simple way to create a standard user interface to color. It takes a string specification of a color, typically from a command line or XGetDefault option, and returns the corresponding red, green, and blue values that are suitable for a subsequent call to XAllocColor or XStoreColor. The color can be specified either as a color name (as in XAllocNamedColor) or as 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 (both uppercase and lowercase). When fewer than 16 bits each are specified, they represent the most-significant bits of the value. For example, #3a7 is the same as #3000a0007000. The colormap is used only to determine which screen to look up the color on. For example, you can use the screen's default colormap.

If the initial character is a sharp sign but the string otherwise fails to fit the above formats or if the initial character is not a sharp sign and the named color does not exist in the server's database, `XParseColor` fails and returns zero.

`XParseColor` can generate a `BadColor` error.

10.5 Generating Regions

Regions are arbitrary sets of pixel locations. Xlib provides functions for manipulating regions. The opaque type `Region` is defined in `<X11/Xutil.h>`.

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 generate the smallest rectangle enclosing the 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.

10.6 Manipulating Regions

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

10.6.1 Creating, Copying, or Destroying Regions

To create a new empty region, use `XCreateRegion`.

```
Region XCreateRegion()
```

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.

10.6.2 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.

10.6.3 Computing with Regions

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.

10.6.4 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.

10.6.5 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.

10.7 Using the Cut and Paste Buffers

Xlib provides functions that you can use to cut and paste buffers for programs using this form of communications. Selections are a more useful mechanism for interchanging data between clients because typed information can be exchanged. X provides property names for properties in which bytes can be stored for implementing cut and paste between windows (implemented by use of properties on the first root window of the display). It is up to applications to agree on how to represent the data in the buffers. The data is most often ISO Latin-1 text. The atoms for eight such buffer names are provided and can be accessed as a ring or as explicit buffers (numbered 0 through 7). New applications are encouraged to share data by using selections (see section 4.4).

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 cut buffer's contents need not be text, so zero bytes are not special. 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 the property for the buffer has never been created, a `BadAtom` error results.

XStoreBuffer can generate BadAlloc and BadAtom errors.

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. Note that the cut buffer does not necessarily contain text, so it may contain embedded zero bytes and may not terminate with a null byte.

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.

XFetchBuffer can generate a BadValue error.

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.

10.8 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 match 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.

10.9 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 forth) 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 6.7.

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 a pointer to the visual.
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 a pointer to 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 offset to a Z format 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 a pointer to 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.

```
int XPutPixel(ximage, x, y, pixel)
      XImage *ximage;
      int x;
      int y;
      unsigned long pixel;
```

ximage Specifies a pointer to 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 a pointer to 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 the pixmap by a constant value, use `XAddPixel`.

```
XAddPixel(ximage, value)
      XImage *ximage;
      long value;
```

ximage Specifies a pointer to 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`.

```
int XDestroyImage(ximage)
        XImage *ximage;
```

ximage Specifies a pointer to 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.

10.10 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 char name_bits[] = { 0xNN,... }
```

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 other variables are required. 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 can be either in the standard X version 10 format (that is, the format used by X version 10 bitmap program) or in the X version 11 bitmap format. 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. While `XReadBitmapFile` can read in either X version 10 format or X version 11 format, `XWriteBitmapFile` always writes out X version 11 format. 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.

10.11 Using the Resource Manager

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. Presuming that all applications use the resource manager, a user can define this information using only two lines of specifications. Your personal resource database usually is stored in a file and is loaded onto a server property when you log in. This database is retrieved automatically by Xlib when a connection is opened.

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 lowercase. 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 retrieve (include) new mail and has the name "include". This window has a fully qualified name, "xmh.toc.buttons.include", and a fully qualified class, "Xmh.VPaned.Box.Command". Its fully qualified name is the name of its parent, "xmh.toc.buttons", followed by its name, "include". Its class is the class of its parent, "Xmh.VPaned.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.

This include button needs the following resources:

- Title string
- Font
- Foreground color for its inactive state
- Background color for its inactive state
- Foreground color for its active state
- Background color for its active state

Each of the resources that this button needs are considered to be attributes of the button and, as such, have a name and a class. For example, the foreground color for the button in its active state might be named "activeForeground", and its class would 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 look-up routine. After look up, the resource manager returns the resource value and the representation type.

The resource manager allows applications to store resources by an incomplete specification of name, class, and a representation type, as well as to retrieve them given a fully qualified name and class.

10.11.1 Resource Manager Matching Rules

The algorithm for determining which resource name or names match a given query is the heart of the database. Resources are stored with only partially specified names and classes, using pattern matching constructs. An asterisk (*) is used to represent any number of intervening components (including none). A period (.) is used to separate immediately adjacent components. All queries fully specify the name and class of the resource needed. A trailing period and asterisk are not removed. The library supports 100 components in a name or class. The look-up algorithm then searches the database for the name that most closely matches (is most specific) this full name and class. The rules for a match in order of precedence are:

1. The attribute of the name and class must match. For example, queries for:

```
xterm.scrollbar.background    (name)
XTerm.Scrollbar.Background    (class)
```

will not match the following database entry:

```
xterm.scrollbar:on
```

2. Database entries with name or class prefixed by a period (.) are more specific than those prefixed by an asterisk (*). For example, the entry xterm.geometry is more specific than the entry xterm*geometry.
3. Names are more specific than classes. For example, the entry "*scrollbar.background" is more specific than the entry "*Scrollbar.Background".
4. Specifying a name or class is more specific than omitting either. For example, the entry "Scrollbar*Background" is more specific than the entry "*Background".
5. Left components are more specific than right components. For example, "*vt100*background" is more specific than the entry "*scrollbar*background" for the query ".vt100.scrollbar.background".
6. If neither a period (.) nor an asterisk (*) is specified at the beginning, a period (.) is implicit. For example, "xterm.background" is identical to ".xterm.background".

Names and classes can be mixed. As an example of these rules, assume the following user preference specification:

```
xmh*background:              red
*command.font:               8x13
*command.background:         blue
*Command.Foreground:         green
xmh.toc*Command.activeForeground:black
```

A query for the name "xmh.toc.messagefunctions.include.activeForeground" and class "Xmh.VPaned.Box.Command.Foreground" would match "xmh.toc*Command.activeForeground" and return "black". However, it also matches "*Command.Foreground".

Using the precedence algorithm described above, the resource manager would return the value specified by "xmh.toc*Command.activeForeground".

10.11.2 Basic Resource Manager Definitions

The definitions for the resource manager's use are contained in `<X11/Xresource.h>`. Xlib also uses the resource manager internally to allow for non-English language error messages.

Database values consist 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 contains:

```
typedef struct {
      unsigned int size;
      caddr_t addr;
} XrmValue, *XrmValuePtr;
```

A resource database is an opaque type used by the look-up functions.

```
typedef struct _XrmHashBucketRec *XrmDatabase;
```

To initialize the resource manager, use XrmInitialize.

```
void XrmInitialize();
```

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 guaranteed not to represent any string that is known to the resource manager.

To allocate some memory you will never give back, use Xpermalloc.

```
char *Xpermalloc(size)
     unsigned int size;
```

The Xpermalloc function is used by some toolkits for permanently allocated storage and allows some performance and space savings over the completely general memory allocator.

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;
```

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`.

```
#define XrmStringToName(string) XrmStringToQuark(string)
#define XrmStringToClass(string) XrmStringToQuark(string)
#define XrmStringToRepresentation(string) XrmStringToQuark(string)
```

```
XrmQuark XrmStringToQuark(string)
     char *string;
```

string Specifies the string for which a quark is to be allocated.

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.

These functions can be used to convert to and from quark representations. The string pointed to by the return value must not be modified or freed. If no string exists for that quark, `XrmQuarkToString` returns NULL.

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 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. The components of the string are separated by a period or asterisk character.

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 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. If the string does not start with a period or an asterisk, a period is assumed. For example, "*a.b*c" becomes:

```
quarks   a      b      c
bindings loose  tight  loose
```

10.11.3 Resource Database Access

Xlib provides resource management functions that you can use to manipulate resource databases. The next sections discuss how to:

- Store and get resources

- Get database levels
- Merge two databases
- Retrieve and store databases

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 a pointer to 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)
```

```
void XrmQPutResource(database, bindings, quarks, type, value)
     XrmDatabase *database;
     XrmBindingList bindings;
     XrmQuarkList quarks;
     XrmRepresentation type;
     XrmValue *value;
```

database Specifies a pointer to 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.

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 a pointer to 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 routine that takes both the resource and value as null-terminated strings, converts them to quarks, and then calls `XrmQPutResource`, using a "String" representation type.

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 a pointer to 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.

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 a pointer to the resource database.

line Specifies the resource value pair as a single string. A single colon (:) separates the name from the value.

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. Any white space before or after the name or colon in the line argument is ignored. The value is terminated by a new-line or a NULL character. To allow values to contain embedded new-line characters, a "\n" is recognized and replaced by a new-line character. For example, line might have the value "xterm*background:green\n". Null-terminated strings without a new line are also permitted.

Looking Up from a Resource Database

To retrieve a resource from a resource database, use `XrmGetResource` or `XrmQGetResource`.

```
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 a pointer to 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 a pointer to 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`.

Database Search Lists

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 return 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.

Merging Resource Databases

To merge the contents of one database into another database, 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 a pointer to the resource database into which the source database is to be merged.

The `XrmMergeDatabases` function merges the contents of one database into another. It may overwrite entries in the destination database. This function is used to combine databases (for example, an application specific database of defaults and a database of user preferences). The merge is destructive; that is, the source database is destroyed.

Retrieving and Storing Databases

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 lines in the format accepted by `XrmPutLineResource`. 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. The file is an ASCII text file that contains lines in the format that is accepted by `XrmPutLineResource`.

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. Each line is separated by a new-line character in the format accepted by XrmPutLineResource.

10.11.4 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 OptionDescRec.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 */
} XrmOptionKind;

typedef struct {
      char *option;            /* Option specification string in argv    */
      char *resourceName;      /* Binding and resource name (sans application name)    */
      XrmOptionKind argKind;   /* Which style of option it is    */
      caddr_t value;           /* Value to provide if XrmoptionNoArg    */
} 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 a pointer to 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 a pointer to 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.

The specified table is used to parse the command line. Recognized entries in the table are removed from argv, and entries are made in 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 argc argument specifies the number of arguments in argv and is set 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 the specification. No separating (binding) character is inserted. 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.

For example, the following is part of the standard option table from the X Toolkit XtInitialize function:

```
static XrmOptionDescRec opTable[] = {
{"-background",  "*background",                XrmoptionSepArg,(caddr_t) NULL},
{"-bd",          "*borderColor",               XrmoptionSepArg,(caddr_t) NULL},
{"-bg",          "*background",                XrmoptionSepArg,(caddr_t) NULL},
{"-borderwidth", "*TopLevelShell.borderWidth", XrmoptionSepArg,(caddr_t) NULL},
{"-bordercolor", "*borderColor",               XrmoptionSepArg,(caddr_t) NULL},
{"-bw",          "*TopLevelShell.borderWidth", XrmoptionSepArg,(caddr_t) NULL},
{"-display",     ".display",                   XrmoptionSepArg,(caddr_t) NULL},
{"-fg",          "*foreground",                XrmoptionSepArg,(caddr_t) NULL},
{"-fn",          "*font",                      XrmoptionSepArg,(caddr_t) NULL},
{"-font",        "*font",                      XrmoptionSepArg,(caddr_t) NULL},
{"-foreground",  "*foreground",                XrmoptionSepArg,(caddr_t) NULL},
{"-geometry",    ".TopLevelShell.geometry",    XrmoptionSepArg,(caddr_t) NULL},
{"-iconic",      ".TopLevelShell.iconic",      XrmoptionNoArg,(caddr_t) "on"},
{"-name",        ".name",                      XrmoptionSepArg,(caddr_t) NULL},
{"-reverse",     "*reverseVideo",              XrmoptionNoArg,(caddr_t) "on"},
{"-rv",          "*reverseVideo",              XrmoptionNoArg,(caddr_t) "on"},
{"-synchronous", ".synchronous",               XrmoptionNoArg,(caddr_t) "on"},
{"-title",       ".TopLevelShell.title",       XrmoptionSepArg,(caddr_t) NULL},
{"-xrm",         NULL,                         XrmoptionResArg,(caddr_t) 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 uppercase and lowercase matter.

10.12 Using the Context Manager

The context manager provides a way of associating data with 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 window, and each piece of data has a type associated with it. The context manager requires knowledge of the window 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 window 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 window and context type, use `XSaveContext`.

```
int XSaveContext(display, w, context, data)
      Display *display;
      Window w;
      XContext context;
      caddr_t data;
```

display Specifies the connection to the X server.

w Specifies the window 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 window 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 window and type, use `XFindContext`.

```
int XFindContext(display, w, context, data_return)
      Display *display;
      Window w;
      XContext context;
      caddr_t *data_return;
```

display	Specifies the connection to the X server.
w	Specifies the window with which the data is associated.
context	Specifies the context type to which the data belongs.
data_return	Returns a pointer to 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 window and type, use `XDeleteContext`.

```
int XDeleteContext(display, w, context)
      Display *display;
      Window w;
      XContext context;
```

display	Specifies the connection to the X server.
w	Specifies the window 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 window 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()
```

Xlib Functions and Protocol Requests A

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
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
XGetFontPath	GetFontPath
XGetGeometry	GetGeometry
XGetIconSizes	GetProperty
XGetImage	GetImage
XGetInputFocus	GetInputFocus
XGetKeyboardControl	GetKeyboardControl
XGetKeyboardMapping	GetKeyboardMapping
XGetModifierMapping	GetModifierMapping
XGetMotionEvents	GetMotionEvents
XGetModifierMapping	GetModifierMapping
XGetNormalHints	GetProperty
XGetPointerControl	GetPointerControl
XGetPointerMapping	GetPointerMapping
XGetScreenSaver	GetScreenSaver
XGetSelectionOwner	GetSelectionOwner
XGetSizeHints	GetProperty
XGetWMHints	GetProperty
XGetWindowAttributes	GetWindowAttributes
	GetGeometry
XGetWindowProperty	GetProperty
XGetZoomHints	GetProperty
XGrabButton	GrabButton
XGrabKey	GrabKey
XGrabKeyboard	GrabKeyboard
XGrabPointer	GrabPointer
XGrabServer	GrabServer
XInitExtension	QueryExtension
XInstallColormap	InstallColormap
XInternAtom	InternAtom
XKillClient	KillClient
XListExtensions	ListExtensions

XListFonts	ListFonts
XListFontsWithInfo	ListFontsWithInfo
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
XRemoveFromSaveSet	ChangeSaveSet
XRemoveHost	ChangeHosts
XRemoveHosts	ChangeHosts
XReparentWindow	ReparentWindow

XResetScreenSaver	ForceScreenSaver
XResizeWindow	ConfigureWindow
XRestackWindows	ConfigureWindow
XRotateBuffers	RotateProperties
XRotateWindowProperties	RotateProperties
XSelectInput	ChangeWindowAttributes
XSendEvent	SendEvent
XSetAccessControl	SetAccessControl
XSetArcMode	ChangeGC
XSetBackground	ChangeGC
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
XSetScreenSaver	SetScreenSaver
XSetSelectionOwner	SetSelectionOwner
XSetSizeHints	ChangeProperty
XSetStandardProperties	ChangeProperty
XSetState	ChangeGC
XSetStipple	ChangeGC
XSetSubwindowMode	ChangeGC
XSetTile	ChangeGC
XSetTSOrigin	ChangeGC
XSetWMHints	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
XTranslateCoordinates	TranslateCoordinates
XUndefineCursor	ChangeWindowAttributes
XUngrabButton	UngrabButton
XUngrabKey	UngrabKey
XUngrabKeyboard	UngrabKeyboard
XUngrabPointer	UngrabPointer
XUngrabServer	UngrabServer
XUninstallColormap	UninstallColormap
XUnloadFont	CloseFont
XUnmapSubwindows	UnmapSubwindows
XUnmapWindow	UnmapWindow
XWarpPointer	WarpPointer

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
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
	XSetCommand
	XSetIconName
	XSetIconSizes
	XSetNormalHints
	XSetSizeHints
	XSetStandardProperties
	XSetWMHints
	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
	XResizeWindow
	XRestackWindows
	XSetWindowBorderWidth
ConvertSelection	XConvertSelection
CopyArea	XCopyArea

CopyColormapAndFree	XCopyColormapAndFree
CopyGC	XCopyGC
CopyPlane	XCopyPlane
CreateColormap	XCreateColormap
CreateCursor	XCreatePixmapCursor
CreateGC	XCreateGC
	XCreateBitmapFromData
	XCreatePixmapFromData
	XOpenDisplay
	XReadBitmapFile
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
GetKeyboardControl	XGetKeyboardControl
GetKeyboardMapping	XGetKeyboardMapping
GetModifierMapping	XGetModifierMapping

GetMotionEvents	XGetMotionEvents
GetPointerControl	XGetPointerControl
GetPointerMapping	XGetPointerMapping
GetProperty	XFetchBytes
	XFetchName
	XGetIconSizes
	XGetNormalHints
	XGetSizeHints
	XGetWMHints
	XGetWindowProperty
	XGetZoomHints
GetSelectionOwner	XGetSelectionOwner
GetWindowAttributes	XGetWindowAttributes
GrabButton	XGrabButton
GrabKey	XGrabKey
GrabKeyboard	XGrabKeyboard
GrabPointer	XGrabPointer
GrabServer	XGrabServer
ImageText16	XDrawImageString16
ImageText8	XDrawImageString
InstallColormap	XInstallColormap
InternAtom	XInternAtom
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

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	XSendEvent
SetClipRectangles	XSetClipRectangles
SetCloseDownMode	XSetCloseDownMode
SetDashes	XSetDashes
SetFontPath	XSetFontPath
SetInputFocus	XSetInputFocus
SetModifierMapping	XSetModifierMapping
SetPointerMapping	XSetPointerMapping
SetScreenSaver	XGetScreenSaver

	XSetScreenSaver
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
WarpPointer	XWarpPointer

Xlib Font Cursors

B

The following are the available cursors that can be used with `XCreateFontCursor`.

```
#define XC_X_cursor 0
#define XC_arrow 2
#define XC_based_arrow_down 4
#define XC_based_arrow_up 6
#define XC_boat 8
#define XC_bogosity 10
#define XC_bottom_left_corner 12
#define XC_bottom_right_corner 14
#define XC_bottom_side 16
#define XC_bottom_tee 18
#define XC_box_spiral 20
#define XC_center_ptr 22
#define XC_circle 24
#define XC_clock 26
#define XC_coffee_mug 28
#define XC_cross 30
#define XC_cross_reverse 32
#define XC_crosshair 34
#define XC_diamond_cross 36
#define XC_dot 38
#define XC_dot_box_mask 40
#define XC_double_arrow 42
#define XC_draft_large 44
#define XC_draft_small 46
#define XC_draped_box 48
#define XC_exchange 50
#define XC_fleur 52
#define XC_gobbler 54
#define XC_gumby 56
#define XC_hand 58
#define XC_hand1_mask 60
#define XC_heart 62
#define XC_icon 64
#define XC_iron_cross 66
#define XC_left_ptr 68
#define XC_left_side 70
#define XC_left_tee 72
#define XC_leftbutton 74
#define XC_ll_angle 76
#define XC_lr_angle 78
#define XC_man 80
#define XC_middlebutton 82
#define XC_mouse 84
#define XC_pencil 86
#define XC_pirate 88
#define XC_plus 90
#define XC_question_arrow 92
#define XC_right_ptr 94
#define XC_right_side 96
#define XC_right_tee 98
#define XC_rightbutton 100
#define XC_rtl_logo 102
#define XC_sailboat 104
#define XC_sb_down_arrow 106
#define XC_sb_h_double_arrow 108
#define XC_sb_left_arrow 110
#define XC_sb_right_arrow 112
#define XC_sb_up_arrow 114
#define XC_sb_v_double_arrow 116
#define XC_shuttle 118
#define XC_sizing 120
#define XC_spider 122
#define XC_spraycan 124
#define XC_star 126
#define XC_target 128
#define XC_tcross 130
#define XC_top_left_arrow 132
#define XC_top_left_corner 134
#define XC_top_right_corner 136
#define XC_top_side 138
#define XC_top_tee 140
#define XC_trek 142
#define XC_ul_angle 144
#define XC_umbrella 146
#define XC_ur_angle 148
#define XC_watch 150
#define XC_xterm 152
```

Extensions C

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 favorite 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>`.

C.1 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;
```

XQueryExtension determines if the named extension is present. If so, the major opcode for the extension is returned (if it has one); otherwise, False 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; otherwise, False 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; otherwise, False is returned. The format of additional data in the errors is specific to the extension.

The extension name should be in the ISO Latin-1 encoding, and uppercase and lowercase do matter.

```
char **XListExtensions(display, nextensions_return)
      Display *display;
      int *nextensions_return;
```

XListExtensions returns a list of all extensions supported by the server.

```
XFreeExtensionList(list)
      char **list;
```

XFreeExtensionList frees the memory allocated by XListExtensions.

C.2 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;
```

`XInitExtension` determines if the 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.

In particular, 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;
```

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.)

C.3 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 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;            /* display */
      int extension;               /* extension number */
      int (*proc)();               /* routine to call when display 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;               /* display */
      int extension;                  /* extension number */
      int (*proc)();                  /* routine to call when GC created */
```

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;               /* display */
      int extension;                  /* extension number */
      int (*proc)();                  /* routine to call when GC 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;               /* display */
      int extension;                  /* extension number */
      int (*proc)();                  /* routine to call when GC 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;               /* display */
      int extension;                  /* extension number */
      int (*proc)();                  /* routine to call when font 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;               /* display */
      int extension;                  /* extension number */
      int (*proc)();                  /* routine to call when font 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.

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;              /* display */
      int event_number;              /* event routine to replace */
      Bool (*proc)();                /* routine to call when converting 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, type must be the first member and window must be the second member. 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.

```
Status (*XESetEventToWire(display, event_number, proc))()
      Display *display;               /* display */
      int event_number;               /* event routine to replace */
      int (*proc)();                  /* routine to call when converting 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 nonzero 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 wire format to host 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. In the XEvent structure that you are forming, you must have "type" as the first member and "window" as the second. You then should fill in the type with the type from the xEvent structure. All other members then should be copied from the wire format to the XEvent structure.

```
int (*XESetError(display, extension, proc))()
      Display *display;               /* display */
      int extension;                  /* extension number */
      int (*proc)();                  /* routine to call when X error happens */
```

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 8.12.2.) If your routine returns nonzero, the error is suppressed, and _XReply returns the value of ret_code.

```
char  *(*XESetErrorString(display, extension, proc))()
      Display *display;              /* display */
      int extension;                 /* extension number */
      char *(*proc)();               /* 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.

```
int (*XESetFlushGC(display, extension, proc))()
      Display *display;              /* display */
      int extension;                 /* extension number */
      char *(*proc)();               /* routine to call when I/O error happens */
```

The XESetFlushGC procedure is identical to XESetCopyGC except that XESetFlushGC is called when a GC cache needs to be updated in the server.

C.4 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)();              /* if defined,  called to free private */
      char *private;              /* 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 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;
```

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)
      struct _XExtData **structure;/* pointer to structure to add */
      XExtData *ext_data;/* 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;/* 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;
```

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.

C.5 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 behavior 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;
```

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;
```

C.6 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);
        }
    point->x = x;
    point->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.

C.7 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 "X Window System Protocol", available from MIT.

C.8 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.

C.9 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.

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

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. Extensions should be designed in such a way that long protocol requests can be split up into smaller requests. The protocol guarantees the size to be no smaller than 4096 unit (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 nonzero 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).

C.10 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;
```

C.11 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;
```

C.12 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);
```

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);
```

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);
/* 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.

C.13 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);

Data32(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.

C.14 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;                    /* number of 32-bit words expected after the reply */
      Bool discard;                 /* should I discard data following "extra" words? */
```

`_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 bytes (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, nbytes)
       Display *display;
       char *data;
       long nbytes;
```

_XRead reads the specified number of bytes into data.

```
_XRead16(display, data, nbytes)
      Display *display;
      short *data;
      long nbytes;
```

_XRead16 reads the specified number of bytes, unpacking them as 16-bit quanities, into the specified array as shorts.

```
_XRead32(display, data, nbytes)
      Display *display;
      long *data;
      long nbytes;
```

_XRead32 reads the specified number of bytes, unpacking them as 32-bit quanities, into the specified array as longs.

```
_XRead16Pad(display, data, nbytes)
      Display *display;
      short *data;
      long nbytes;
```

_XRead16Pad reads the specified number of bytes, unpacking them as 16-bit quanities, into the specified array as shorts. If the number of bytes is not a multiple of four, _XRead16Pad reads up to three additional pad bytes.

```
_XReadPad(display, data, nbytes)
      Display *display;
      char *data;
      long nbytes;
```

_XReadPad reads the specified number of bytes into data. If the number of bytes is not a multiple of four, _XReadPad reads up to three additional pad bytes.

Each protocol request is a little different. For further information, see the Xlib sources for examples.

C.15 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.

C.16 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;
```

This storage must only be used inside of the critical section of your stub.

C.17 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 nonnegative 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.

C.18 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.

Version 10 Compatibility Functions D

D.1 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 X10 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 X10 functions `XDraw`, `XDrawFilled`, `XDrawPatterned`, `XDrawDashed`, and `XDrawTiled`. They are as compatible as possible given X11'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 X10 (this is the X11 standard error status). `XAppendVertex` and `XClearVertexFlag` from X10 also are not supported.

The setup of the graphics context determines whether you get dashes, 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.

`XDraw` achieves the effects of X10 `XDrawDashed`, and `XDrawPatterned`.

```
#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.

XDrawTiled achieves the effects of X10 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.

D.2 Associating User Data with a Value

These functions are superseded by the context management functions (see section 10.12). It is often necessary to associate arbitrary information with resource IDs. Xlib provides the `XAssocTable` functions used in making such an association. Application programs often must 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`, etc.).

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.

XAssocTable Frees memory associated with a given XDestroyAssocTable.

```
XDestroyAssocTable(table)
      XAssocTable *table;
```

table Specifies the assoc table.

HP Extensions E

To provide better integration with existing products and peripherals available with HP 9000 computers, a number of extensions have been added to the X Window System. These extensions add to the existing X standard, creating a superset of functionality. These features will work among all networked HP 9000 computers, but may not work with other vendor's systems on the same network.

E.1 Input Device Extensions

The standard input model for X consists of a keyboard and a mouse. The actual devices used may be something other than a keyboard or mouse, but the model assumes that one device has keys and is treated like a keyboard and the other is a pointer that is treated like a mouse. This input model meets the needs of most users and is what standard X client programs expect.

This standard model of input has some limitations. For example, it does not provide a way to easily use multiple input devices at the same time. In addition, in some applications a mouse may not be the appropriate input device.

To meet this need and provide greater flexibility in the use of HP-HIL input devices with X, an extended set of input features have been built into the X server and an extended features library called *libXhp11.a*. A programmatic interface is provided that can be used by new or modified client programs.

None of these features are required in order for the X server or X clients to operate correctly if only the standard input devices are desired. They are provided as extensions to the capabilities of X that may be used in addition to the standard input features.

By default, the X server uses a mouse as the pointer device and a keyboard as its key device (if they are attached). For information specifying other devices as the X pointer and keyboard, refer to *Using the X Window System* (HP Part Number 98794-90001).

E.1.1 Programming with Extended Input

Existing client programs may be modified, or new client programs may be written to take advantage of the extended input functions. These functions allow client programs to determine what input devices are available, determine information about each device, and access individual devices.

E.1.2 Listing Available Devices

To obtain a list of available input devices, use `XHPListInputDevices`.

```
XHPDeviceList *XHPListInputDevices(display, ndevices)
      Display *display;
      int *ndevices;        /* RETURN */
```

display Specifies the connection to the X server.

ndevices Specifies as a return value the number of devices available.

`XHPListInputDevices` returns information about the input devices that are available to the X server, including the standard X keyboard and pointer devices. Each time it is called it returns a pointer to an array of `XHPDeviceList` structures that contains information about each device. The `ndevices` value returned specifies the number of `XHPDeviceList` structures in the array. In `<X11/XHPlib.h>`, the `XHPDeviceList` structure is defined as follows:

```
typedef struct
      {
      unsigned int      resolution;  /* resolution in counts/ meter*/
      unsigned short    min_val;     /* min value this axis returns*/
      unsigned short    max_val;     /* max value this axis returns*/
      } XHPaxis_info;

typedef struct
      {
      XID     x_id;                   /* device X identifier        */
      char    *name;                  /* device name                */
      XHPaxis_info      *axes;        /* pointer to axes array      */
      unsigned short    type;         /* device type                */
      unsigned short    min_keycode;  /* min X keycode from this dev*/
      unsigned short    max_keycode;  /* max X keycode from this dev*/
      unsigned char     hil_id;       /* device HIL identifier      */
      unsigned char     mode;         /* ABSOLUTE or RELATIVE       */
      unsigned char     num_axes;     /* # axes this device has     */
      unsigned char     num_buttons;  /* # buttons on this device   */
      unsigned char     num_keys;     /* # keys on this device      */
      unsigned char     io_byte;      /* I/O descriptor byte for dev*/
      unsigned char     pad[8]        /* reserved for future use    */
      } XHPDeviceList;
```

The axes field of the HPDeviceList structure contains the address of an array of XHPaxis_info structures. The num_axes field contains the number of elements in this array. If the num_axes field contains 0 (zero), the contents of the axes field will be NULL. In the XHPaxis_info structure the resolution field contains the resolution of the device in counts per meter. If the mode field of the XHPDeviceList structure is ABSOLUTE, then the min_val and max_val fields contain the minimum and maximum values the device can report. For relative pointing devices, these fields contain 0 (zero).

The X pointer device is always the first device listed and has an x_id field equal to the constant XPOINTER. The X keyboard device is always listed second and has an x_id field equal to the constant XKEYBOARD. In general, attempting to access the X keyboard or pointer devices using the HP extension functions generates a BadDevice error.

A variety of device types are defined in <X11/XHPlib.h>.

Name	Device Type
MOUSE	HP-HIL mouse
TABLET	HP-HIL graphics tablet
KEYBOARD	HP-HIL keyboard
TOUCHSCREEN	HP-HIL touchscreen
TOUCHPAD	HP-HIL touchpad
BUTTONBOX	HP-HIL buttonbox
BARCODE	HP-HIL barcode reader
ONE_KNOB	HP-HIL single knob box
NINE_KNOB	HP-HIL nine knob box
TRACKBALL	HP-HIL trackball
QUADRATURE	HP-HIL quadrature

XHPDeviceList returns NULL if there are no input devices to list.

E.1.3 Freeing the DeviceList

To free an XHPDeviceList array created by XHPListInputDevices, use XHPFreeDeviceList.

```
void XHPFreeDeviceList(list)
     XHPDeviceList *list;
```

list Specifies the XHPDeviceList to free.

When XHPListInputDevices is called it allocates memory to place the XHPDeviceList array into. To free this allocated memory call XHPFreeDeviceList with the XHPDeviceList list pointer as an argument. This frees the memory previously allocated.

E.1.4 Enabling Extended Input Devices

To enable an extended input device, use XHPSetInputDevice.

```
int XHPSetInputDevice(display,deviceid,mode)
      Display *display;
      XID deviceid;
      int mode;
```

display Specifies the connection to the X server.

deviceid Specifies the device to open or close. This is a deviceid listed in the XHPDeviceList structure.

mode Controls the mode to which the device is set (ON|SYSTEM_EVENTS, ON|DEVICE_EVENTS, or OFF).

XHPSetInputDevice allows a client program to request the server to open a device or to close a device when it is no longer needed. The client may cause the device to be treated as an extension of the X keyboard or X pointer by using the mode SYSTEM_EVENTS, or as an individually-selectable device by using the mode DEVICE_EVENTS. Valid values for the mode parameter are ON|SYSTEM_EVENTS, ON|DEVICE_EVENTS, or OFF.

Most clients will want to use DEVICE_EVENTS so that the events generated by an extended input device can be distinguished from those generated by the X keyboard and pointer devices.

XHPSetInputDevice may return BadDevice or BadMode errors. A BadMode error is generated if another client has opened the device with a conflicting mode.

E.1.5 Getting the Event Select Mask and Event Type

Event masks and event types for the events returned by extended input devices are not constants. Instead, they are allocated by the X server during its initialization. Therefore, client programs must request from the server the event masks to be used to select extended input *and* the event types to be compared with an event when it is received.

To obtain an event mask and event type for a specific extended input event, use XHPGetExtEventMask.

```
int XHPGetExtEventMask(display,event_constant,eventtype,mask)
      Display *display;
      long event_constant;
      long *eventtype;      /* RETURN */
      long *mask;             /* RETURN */
```

display	Specifies the connection to the X server.
event_constant	Specifies the constant corresponding to the extended event you wish to receive.
eventtype	Address of a variable into which the server can return the event type for the extended input event.
mask	Address of a variable into which the server can return the event mask to use in selecting that event.

The client program must request the event mask and event type to be used in selecting the events returned by devices. It does this by calling the server with a constant that corresponds to the desired event. The server returns the event mask and event type for the desired event. Valid constants that may be used by the client to request corresponding event masks and types are shown in the following table:

Mask Request	Description
HPDeviceKeyPressreq	Request HPDeviceKeyPress event mask and event type for a extended device.
HPDeviceKeyReleasereq	Request HPDeviceKeyRelease event mask and event type for an extended device.
HPDeviceButtonPressreq	Request HPDeviceButtonPress event mask and event type for an extended device.
HPDeviceButtonReleasereq	Request HPDeviceButtonRelease event mask and event type for an extended device.
HPDeviceMotionNotifyreq	Request HPDeviceMotionNotify event mask and event type for an extended device.
HPDeviceFocusInreq	Request HPDeviceFocusIn event mask and event type for an extended device.
HPDeviceFocusOutreq	Request HPDeviceFocusOut event mask and event type for an extended device.
HPProximityInreq	Request HPProximityIn event mask and event type for an extended device.
HPProximityOutreq	Request HPProximityOut event mask and event type for an extended device.
HPDeviceKeymapNotifyreq	Request HPDeviceKeymapNotify event mask and event type for an extended device.
HPDeviceMappingNotifyreq	Request HPDeviceMapping event type for an extended device. (There is no event mask for this event.)

XHPGetExtMask may return a BadType error.

E.1.6 Selecting Input From Extended Input Devices

To select input from an extended input device, use XHPSelectExtensionEvent.

```
XHPSelectExtensionEvent(display, window, deviceid, mask)
      Display *display;
      Window window;
      XID deviceid;
      Mask mask;
```

display Specifies the connection to the X server.

window Specifies the window ID. Client applications interested in an event for a particular window pass that window's ID.

deviceid Specifies the device from which input is desired.

mask Specifies the mask of input events.

The `XHPSelectExtensionEvent` function is provided to support the use of input devices other than the X keyboard and X pointer device. It allows input from extended input devices, selected independently of those events generated by the X pointer and keyboard.

`XHPSelectExtensionEvent` requests that the server send an extended event that matches the specified event mask and is issued from the specified device and window. To use this function, the client program must first determine the appropriate `deviceid` by using the `XHPListInputDevice` function, and the appropriate event mask by using the `XHPGetExtEventMask` function. Multiple event masks returned by `XHPGetExtEventMask` may be OR'd together and specified in a single request to `XHPSelectExtensionEvent`.

`XHPSelectExtensionEvent` cannot be used to select any of the core X events, or to receive input from the X pointer or keyboard devices. Use the `XSelectInput` function for that purpose.

`XHPSelectExtensionEvent` may return a `BadDevice` or `BadWindow` errors.

E.1.7 Grabbing Extended Input Devices

To actively grab an extended input device, use `XHPGrabDevice`.

```
XHPGrabDevice(display, deviceid, grab_window, pointer_mode, device_mode, owner_events, time)
      Display *display;
      char deviceid;
      Window grab_window;
      int pointer_mode;
      int device_mode;
      Bool owner_events;
      Time time;
```

display Specifies the connection to the X server.

device_id Specifies the ID of the device to grab.

grab_window Specifies the window ID of the window associated with the extended input device being grabbed.

pointer_mode Specifies the pointer mode. Only the constant `GrabModeAsync` is currently supported.

device_mode Specifies the device mode. Only the constant `GrabModeAsync` is currently supported.

owner_events Specifies a boolean value of `True`or `False`.

time Specifies the time. You can pass either a timestamp, expressed in milliseconds, or `CurrentTime`.

The `XHPGrabDevice` function actively grabs control of the device and generates `HPDeviceFocusIn` and `HPDeviceFocusOut` events. Further device events are reported only to the grabbing client. This function overrides any active input device 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`, then if a generated device event would normally be reported to this client, it is reported normally; otherwise the event is reported with respect to the grab_window. Regardless of any event selection by the client, both `HPDeviceKeyPress` and `HPDeviceKeyRelease` events are always reported.

`XHPGrabDevice` cannot be used to grab the X pointer device or the X keyboard device. The standard `XGrabKeyboard` and `XGrabPointer` functions should be used for that purpose.

`XHPGrabDevice` can generate `BadValue` and `BadWindow` errors.

E.1.8 Ungrabbing Extended Input Devices

To release a previously grabbed extended input device, use `XHPUngrabDevice`.

```
*XHPUngrabDevice(display,deviceid, time)
      Display *display;
      XID deviceid;
      Time time;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the device to grab.

time Specifies the time. You can pass either a timestamp, expressed in milliseconds, or `CurrentTime`.

The `XHPUngrabDevice` function releases the input device. The function does not release the device and any queued events if the specified time is earlier than the last-grab time or is later than the current X server time. It also generates `HPDeviceFocusIn` and `HPDeviceFocusOut` events. If the event window for an active device grab becomes unviewable, the X server automatically performs an `XHPUngrabDevice` request.

`XHPUngrabDevice` can generate a `BadDevice` error.

E.1.9 Grabbing Extended Input Device Buttons

To passively grab a particular button on an extended input device, use `XHPGrabDeviceButton`.

```
XHPGrabDeviceButton(display, deviceid, button, modifiers, grab_window, owner_events,
                    event_mask, pointer_mode, device_mode)
      Display *display;
      XID deviceid;
      unsigned int button;
      unsigned int modifiers;
      Window grab_window;
      Bool owner_events;
      unsigned int event_mask;
      int pointer_mode, device_mode;
```

display	Specifies the connection to the X server.
deviceid	Specifies the ID of the desired device.
button	Specifies the code of the button that is to be grabbed. You can pass either the button or `AnyButton`.
modifiers	Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: `ShiftMask`, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, `Mod5Mask`. You can also pass `AnyModifier`, which is equivalent to issuing the grab request for all possible modifier combinations (including the combination of no modifiers).
grab_window	Specifies the ID of a window associated with the device specified above.
owner_events	Specifies a boolean value of either `True`or `False`.
event_mask	Specifies which device events are to be reported to the client. They can be the bitwise inclusive OR of these device mask bits: `DeviceButtonPressMask`, `DeviceButtonReleaseMask`, `DevicePointerMotionMask`, `DeviceKeymapStateMask`.

pointer_mode Only the constant `GrabModeAsync` is currently supported.

device_mode Only the constant `GrabModeAsync` is currently supported.

`XHPGrabDeviceButton` is provided to support the use of input devices other than the X keyboard and the X pointer device. It allows a client to establish passive grab on a button on an extended input device. That device must have previously been opened (turned on) using `XHPSetInputDevice`.

`XHPGrabDeviceButton` produces a `BadAccess` error if some other client has issued a `XHPGrabDeviceButton` with the same device and button combination on the same window. When using `AnyModifier` or `AnyButton`, the request fails completely and the X server generates a `BadAccess` error and no grabs are established if there is a conflicting grab for any combination.

`XHPGrabDeviceButton` can generate `BadDevice`, `BadAccess`, `BadWindow`, and `BadValue` errors.

This function cannot be used to grab a button on the X pointer device. The core `XGrabButton` function should be used for that purpose.

E.1.10 Ungrabbing Extended Input Device Buttons

To release previously grabbed extended input device buttons, use `XHPUngrabDeviceButton`.

```
XHPUngrabDeviceButton(display, deviceid, button, modifiers, ungrab_window)
      Display *display;
      XID deviceid;
      unsigned int button;
      unsigned int modifiers;
      Window ungrab_window;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

button Specifies the code of the button that is to be ungrabbed. You can pass either the button or `AnyButton`.

modifiers Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: `ShiftMask`, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, `Mod5Mask`. You can also pass `AnyModifier`, which is equivalent to issuing the ungrab request for all possible modifier combinations (including the combination of no modifiers).

ungrab_window Specifies the ID of a window associated with the device specified above.

`XHPUngrabDeviceButton` is provided to support the use of input devices other than the X keyboard and the X pointer device. It allows a client to remove a grab on a button on an extended input device. That device must have previously been opened (turned on) using `XHPSetInputDevice`.

`XHPUngrabDeviceButton` can generate `BadDevice` and `BadWindow` errors.

`XHPUngrabDeviceButton` cannot be used to ungrab a button on the X pointer device. Use the core `XUngrabButton` function for that purpose.

E.1.11 Grabbing Extended Input Device Keys

To passively grab a particular key on an extended input device, use `XHPGrabDeviceButton`.

```
XHPGrabDeviceKey(display, deviceid, keycode, modifiers, grab_window, owner_events,
                 pointer_mode, device_mode)
      Display *display;
      XID deviceid;
      unsigned int button;
      unsigned int modifiers;
      Window grab_window;
      Bool owner_events;
      int pointer_mode, device_mode;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

button Specifies the code of the key that is to be grabbed. You can pass either the button or `AnyKey`.

modifiers Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: `ShiftMask`, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, `Mod5Mask`. You can also pass `AnyModifier`, which is equivalent to issuing the grab request for all possible modifier combinations (including the combination of no modifiers).

grab_window Specifies the ID of a window associated with the device specified above.

owner_events Specifies a boolean value of either `True` or `False`.

event_mask Specifies which device events are to be reported to the client. They can be the bitwise inclusive OR of these device mask bits: DeviceButtonPressMask, DeviceButtonReleaseMask, DevicePointerMotionMask, DeviceKeymapStateMask.

pointer_mode Only the constant GrabModeAsync is currently supported.

device_mode Only the constant GrabModeAsync is currently supported.

XHPGrabDeviceKey is provided to support the use of input devices other than the X keyboard and the X pointer device. It allows a client to establish passive grab on a button on an extended input device. That device must have previously been opened (turned on) using XHPSetInputDevice.

XHPGrabDeviceKey produces a BadAccess error if some other client has issued a XHPGrabDeviceKey with the same device and button combination on the same window. When using AnyModifier or AnyKey, the request fails completely and the X server generates a BadAccess error and no grabs are established if there is a conflicting grab for any combination.

XHPGrabDeviceKey can generate BadDevice, BadAccess, BadWindow, and BadValue errors.

This function cannot be used to grab a key on the X keyboard device. The core XGrabKey function should be used for that purpose.

E.1.12 Ungrabbing Extended Input Device Keys

To release previously grabbed extended input device keys on an extended input device, use XHPUngrabDeviceKey.

```
XHPUngrabDeviceKey(display, deviceid, keycode, modifiers, ungrab_window)
      Display *display;
      XID deviceid;
      unsigned int keycode;
      unsigned int modifiers;
      Window ungrab_window;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

keycode Specifies the code of the key that is to be ungrabbed. You can pass either the key or AnyKey.

modifiers Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: `ShiftMask`, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, `Mod5Mask`. You can also pass `AnyModifier`, which is equivalent to issuing the ungrab request for all possible modifier combinations (including the combination of no modifiers).

ungrab_window Specifies the ID of a window associated with the device specified above.

`XHPUngrabDeviceKey` is provided to support the use of input devices other than the X keyboard and the X pointer device. It allows a client to remove a grab on a key on an extended input device. That device must have previously been opened (turned on) using `XHPSetInputDevice`.

`XHPUngrabDeviceKey` can generate `BadDevice` and `BadWindow` errors.

E.1.13 Getting Extended Input Device Focus

To obtain the focus window id and current focus state of an extended input device, use `XHPGetDeviceFocus`.

```
XHPGetDeviceFocus(display, deviceid, focus_return, revert_to_return)
      Display *display;
      XID deviceid;
      Window *focus_return;          /* RETURN */
      int *revert_to_return;         /* RETURN */
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the device to examine.

focus_return Returns the focus window ID, or either `PointerRoot`, or `None`.

revert_to_return Returns the current focus state. The function can return `RevertToParent`, `RevertToPointerRoot`, or `RevertToNone`.

The `XHPGetDeviceFocus` function returns the focus window ID and the current focus state of the specified extended input device.

E.1.14 Setting Extended Input Device Focus

To set the input focus of an extended input device, use `XHPSetDeviceFocus`.

```
XHPSetDeviceFocus(display, deviceid, focus, revert_to, time)
      Display *display;
      XID deviceid;
      Window focus;
      int revert_to;
      Time time;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the extended device.

focus Specifies the window ID. This is the window in which you want to set the input focus. You can pass a window ID or either `PointerRoot` or `None`.

revert_to Specifies which window 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, expressed in milliseconds, or `CurrentTime`.

The `XHPSetDeviceFocus` function changes the input focus and the last-focus-change time. The function 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). This function causes the X server to generate `XHPDeviceFocusIn` and `XHPDeviceFocusOut` events.

Depending on what value you assign to the focus argument, `XHPSetDeviceFocus` executes as follows:

- If you assign `None` to the focus argument, all device events are discarded until a new focus window is set, and the revert_to argument is ignored.
- If you assign a window ID to the focus argument, it becomes the device's focus window. If a generated device event would normally be reported to this window or one of its inferiors, the event is reported normally. Otherwise, the event is reported relative to the focus window.
- If you assign `PointerRoot` to the focus argument, the focus window is dynamically taken to be the root window of whatever screen the pointer is on at each device event. In this case, the revert_to argument is ignored.

The specified focus window must be viewable at the time `XHPSetDeviceFocus` is called. Otherwise, a `BadMatch` error is generated. If the focus window later becomes not viewable, the X server evaluates the revert_to argument to determine the new focus window:

- If you assign RevertToParent to the revert_to argument, the focus reverts to the parent (or the closest viewable ancestor), and the new revert_to value is taken to be RevertToNone.
- If you assign RevertToPointerRoot or RevertToNone to the revert_to argument, the focus reverts to PointerRoot or None, respectively. The X server generates HPDeviceFocusIn and HPDeviceFocusOut events when the focus reverts, but the last-focus-change time is not affected.

XHPSetDeviceFocus can generate BadMatch, BadValue, BadWindow, and BadDevice errors.

E.1.15 Getting Current Extended Input Event Selection Masks

To obtain the current event selection mask for a specified extended input device and window, use XHPGetCurrentDeviceMask.

```
XHPGetCurrentDeviceMask(display, window, deviceid, mask_return)
      Display *display;
      Window window;
      XID deviceid;
      Mask mask_return;             /* RETURN */
```

display	Specifies the connection to the X server.
window	Specifies the window ID of the window to examine.
deviceid	Specifies the ID of the device to examine.
mask_return	Returns the current extended input event mask.

XHPGetCurrentDeviceMask returns the current event selection mask for the specified extended input device and the specified window. For standard input events, this information is returned by the XGetWindowAttributes function.

XHPGetCurrentDeviceMask can return BadWindow, or BadDevice errors.

E.1.16 Getting Extended Device Motion History

To get the motion history for a specified extended device, window, and time, use XHPGetDeviceMotionEvents.

This function is provided for client programs that need to receive every motion event generated by the X server (such as graphics programs that allow the user to "paint" on the screen). For most other programs, selecting motion events is sufficient. The X server compresses motion events for the X pointer device *and* extended input devices.

```
XHPXTimeCoord *XHPGetDeviceMotionEvents(display, deviceid,
w, start, stop, nevents_return)
      Display *display;
      XID deviceid;
      Window w;
      Time start, stop;
      int *nevents_return;          /* RETURN */
```

display Specifies the connection to the X server.

deviceid Specifies the extended input device.

w Specifies the window ID. The only value currently supported for this parameter is the constant: `ALLWINDOWS`.

start
stop Specify the time interval in which the events are returned from the motion history buffer. You can pass a time stamp, expressed in milliseconds, or `CurrentTime`. If the stop time is in the future, it is equivalent to specifying `CurrentTime`.

nevents_return Returns the number of events from the motion history buffer.

The `XHPGetDeviceMotionEvents` function returns all events in the motion history buffer that fall between the specified start and stop times inclusive. If the start time is later than the stop time or if the start time is in the future, no events are returned. The return type for this function is a structure defined as follows:

```
typedef struct {
      Time time;
      short *data;
} XHPTimeCoord;
```

The time member is set to the time, in milliseconds. The `data` member is a pointer to an array of motion values. The number of elements in this array is determined by the `num_axes` field of the `XHPDeviceList` structure associated the device. You should use `XFree` to free the data returned from this call.

`XHPGetDeviceMotionEvents` can generate a `BadWindow`, or `BadDevice` errors.

E.1.17 Enabling Auto-Repeat for Extended Input Devices

To enable auto-repeat for an extended input device, use `XHPDeviceAutoRepeatOn`.

```
XHPDeviceAutoRepeatOn(display, deviceid, mode)
      Display *display;
      XID deviceid;
      unsigned int mode;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

mode Specifies the auto-repeat rate. Valid values are REPEAT_30, which causes repeats to take place every 1/30th of a second, and REPEAT_60, which causes repeats to take place every 1/60th of a second.

XHPDeviceAutoRepeatOn is provided to support the use of input devices other than the X keyboard and X pointer device. It cannot be used to turn auto-repeat on for the X keyboard device. The core XAutoRepeatOn function should be used for that purpose.

XHPDeviceAutoRepeatOn can generate BadDevice and BadValue errors.

E.1.18 Disabling Auto-Repeat for Extended Input Devices

To disable auto-repeat for an extended input device, use XHPDeviceAutoRepeatOff.

```
XHPDeviceAutoRepeatOff(display, deviceid)
      Display *display;
      XID deviceid;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

XHPDeviceAutoRepeatOff is provided to support the use of input devices other than the X keyboard and X pointer device. It cannot be used to turn auto-repeat off for the X keyboard device. The core XAutoRepeatOff function should be used for that purpose.

XHPDeviceAutoRepeatOff can generate BadDevice and BadValue errors.

E.1.19 Sending a Prompt to Extended Input Devices

To turn on a prompt on an extended input device, use XHPPrompt.

```
XHPPrompt(display, deviceid, prompt)
      Display *display;
      XID deviceid;
      unsigned int prompt;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

prompt Specifies the prompt to be sent. Valid values are: GENERAL_PROMPT, PROMPT_1, PROMPT_2, PROMPT_3, PROMPT_4, PROMPT_5, PROMPT_6, and PROMPT_7.

XHPPrompt sends a prompt to an input device. For example, you can use this function to turn on the prompt light on the HP 46086A 32-button box.

The io_byte field of the XHPDeviceList structure, which is returned by the XHPListInputDevices function, reports which prompts and acknowledges are supported by the device. Bit 7 of the io_byte field corresponds to GENERAL_PROMPT, while bits 6, 5, and 4 are taken as a number between 1 and 7, meaning that prompts numbered 1 through that number are supported.

XHPPrompt can generate BadDevice and BadValue errors.

E.1.20 Sending an Acknowledge to Extended Input Devices

To send an acknowledge signal to an extended input device, use XHPAcknowledge.

```
XHPAcknowledge(display, deviceid, acknowledge)
      Display *display;
      XID deviceid;
      unsigned int acknowledge;
```

display	Specifies the connection to the X server.
deviceid	Specifies the ID of the desired device.
acknowledge	Specifies the acknowledge to be sent. Valid values are: GENERAL_ACKNOWLEDGE, ACKNOWLEDGE_1, ACKNOWLEDGE_2, ACKNOWLEDGE_3, ACKNOWLEDGE_4, ACKNOWLEDGE_5, ACKNOWLEDGE_6, and ACKNOWLEDGE_7.

XHPAcknowledge sends a acknowledge to an input device. For example, you can use this function to turn off the prompt light on the HP 46086A 32-button box.

The io_byte field of the XHPDeviceList structure (returned by the XHPListInputDevices function) reports which prompts and acknowledges are supported by the device. Bit 7 of the io_byte field corresponds to GENERAL_ACKNOWLEDGE, while bits 6, 5, and 4 are taken as a number between 1 and 7, meaning that acknowledges numbered 1 through that number are supported.

XHPAcknowledge can generate BadDevice and BadValue errors.

E.1.21 Getting Control Attributes of Extended Input Devices

To get the control attributes of an extended input device, use XHPGetDeviceControl.

```
XHPGetDeviceControl(display, deviceid, values_return)
      Display *display;
      XID deviceid;
      XHPDeviceState *values_return;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the device whose attributes are to be changed.

values_return Specifies a pointer to the `XHPDeviceState` structure in which the device values will be returned.

`XHPGetDeviceControl` returns the control attributes of input devices (other than the X keyboard and X pointer devices). The specified device must have previously been opened (turned on) with `XHPSetInputDevice`.

`XHPGetDeviceControl` returns the control attributes of the device in the `XHPDeviceState` structure defined as follows:

```
typedef struct {
        int key_click_percent;
        int bell_percent;
        unsigned int bell_pitch;
        unsigned int bell_duration;
        unsigned long led_mask;
        int global_auto_repeat;
        int accelNumerator;
        int accelDenominator;
        int threshold;
        char auto_repeats[32];
} XHPDeviceState;
```

For the LEDs, the lease 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 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. The global_auto_repeat member can be set to either `AutoRepeatModeOn` or `AutoRepeatModeOff`.

This function generates a `BadValue` error if the specified device does not exist, was not previously enabled with `XHPSetInputDevice`, or is the X system pointer or X system keyboard.

E.1.22 Setting Control Attributes of Extended Input Devices

To set control attributes of an extended input device, use `XHPChangeDeviceControl`.

```
XHPChangeDeviceControl(display, deviceid, value_mask, values)
      Display *display;
      XID deviceid;
      unsigned long value_mask;
      XHPDeviceControl *values;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the device whose attributes are to be changed.

value_mask Specifies which attributes are to be changed. Each bit in the mask specifies one attribute of the specified device.

values Specifies a pointer to the `XHPDeviceControl` structure containing the values to be changed.

`XHPChangeDeviceControl` allows the control attributes of input devices (other than the X keyboard and X pointer devices) to be changed. The specified device must have previously been opened (turned on) with `XHPSetInputDevice`.

The attributes to be changed are specified in the `XHPDeviceAttributes` structure. They are not actually changed unless the corresponding bit is set is set in the *value_mask* parameter. The following masks can be ORed into the *value_mask*:

```
#define DVKeyClickPercent       (1L<<0)
#define DVBellPercent           (1L<<1)
#define DVBellPitch             (1L<<2)
#define DVBellDuration          (1L<<3)
#define DVLed                   (1L<<4)
#define DVLedMode               (1L<<5)
#define DVKey                   (1L<<6)
#define DVAutoRepeatMode        (1L<<7)
#define DVAccelNum              (1L<<8)
#define DVAccelDenom            (1L<<9)
#define DVThreshold             (1L<<10)
```

The fields of the `XHPDeviceControl` structure are defined as follows:

```
typedef struct {
        int key_click_percent;
        int bell_percent;
        int bell_pitch;
        int bell_duration;
        int led;
        int led_mode;
        int key;
        int auto_repeat_mode;
        int accelNumerator;
        int accelDenominator;
        int threshold;
} XHPDeviceControl;
```

The key_click_percent and bell_percent members set the volume for key clicks or bell. Allowed values are 0 (off) through 100 (loud). The bell_pitch member sets the pitch (in Hz) of the bell, if possible. The bell_duration member sets the duration (in milliseconds) of the bell, if possible. A value of -1 for any of these members restores the respective default value. Any other negative value generates a `BadValue` error.

If both the led and led_mode 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 an led is specified without an led_mode, a `BadMatch` error is generated.

If both the auto_repeat_mode and key members are specified, the key and auto_repeat_mode 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 device is changed and does not affect the per_key settings. If a key is specified without and auto_repeat_mode, a `BadMatch` error is generated.

E.1.23 Getting the Key Mapping of Extended Input Devices

To get the key mapping of an extended input device, use `XHPGetDeviceKeyMapping`.

```
XHPGetDeviceKeyMapping(display, deviceid, first_keycode_wanted, keycode_count, keysyms_per_keycode_return)
      Display *display;
      XID deviceid;
      KeyCode first_keycode_wanted;
      int keycode_count;
      int keysyms_per_keycode_return;
```

display	Specifies the connection to the X server.
deviceid	Specifies the ID of the device whose keymap is to be returned.
first_keycode_wanted	Specifies the first keycode to be returned.
keycode_count	Specifies the number of keycodes that are to be returned.
keysyms_per_keycode_return	Specifies the number of keysyms per keycode.

`XHPGetDeviceKeyMapping` allows a client program to read and use the key symbols for the keycodes generated by an extended input device (other than the X keyboard and X pointer devices). The specified device must have previously been opened (turned on) with `XHPSetInputDevice`.

Starting with first_keycode_wanted, `XHPGetDeviceKeyMapping` returns the symbols for the specified number of KeyCodes. The specified first_keycode must be greater than or equal to min_keycode supplied at connection setup and stored in the `Display` structure. Also, max_keycode must be greater than first_keycode + keycode_count - 1. If either of these conditions is not met, the function returns a `BadValue` error. The number of elements in the KeySyms list is: keycode_count * keysyms_per_code + N.

KeySym number N, counting from zero, for KeyCode K has the following index in keysyms: (K - first_keycode_wanted) * keysyms_per_keycode_return + N.

The specified keysyms_per_keycode_return 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.

`XHPGetDeviceKeyMapping` can generate `BadDevice` and `BadValue` errors.

E.1.24 Changing the Key Mapping of Extended Input Devices

To change the key mapping of an extended input device, use `XHPChangeDeviceKeyMapping`.

```
XHPChangeDeviceKeyMapping(display, deviceid, first_keycode, keysyms_per_keycode, keysyms, num_codes)
      Display *display;
      XID deviceid;
      int first_keycode;
      int keysyms_per_keycode;
      KeySyms *keysyms;
      int num_codes;
```

display	Specifies the connection to the X server.
deviceid	Specifies the ID of the device whose key map is to be changed.
first_keycode	Specifies the first keycode that is to be changed.
keysyms_per_keycode	Specifies the number of keysyms per keycode.
keysyms	Specifies a pointer to an array of keysyms that are to be used.
num_codes	Specifies the number of keycodes that are to be changed. `XHPDeviceState` structure in which the device values will be returned.

`XHPChangeDeviceKeyMapping` allows a client program to define the key symbols for the keycodes generated by an extended input device (other than the X keyboard and X pointer devices). The specified device must have previously been opened (turned on) with `XHPSetInputDevice`.

Starting with first_keycode, `XHPChangeDeviceKeyMapping` defines the symbols for the specified number of keycodes. The symbols for keycods outside this range remain unchanged. The number of elements must be: num_codes * keysyms_per_keycode. (Otherwise, a `BadLength` error is generated.)

The specified first_keycode must be greater than or equal to min_keycode supplied at connection setup and stored in the `Display` structure. Also, max_keycode must be greater than first_keycode + (num_codes / keysyms_per_keycode) - 1. If either of these conditions is not met, the function returns a `BadValue` error.

KeySym number N, counting from zero, for KeyCode K has the following index in keysyms: (K - first_keycode) * keysyms_per_keycode + N.

The specified keysyms_per_keycode can e 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. `NoSymbol`may a KeyCode. `XHPChangeDeviceKeyMapping` 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.

E.1.25 Setting the Modifier Mapping of Extended Input Devices

To change the modifier mapping of an extended input device, use `XHPSetDeviceModifierMapping`.

```
XHPSetDeviceModifierMapping(display, deviceid, modmap)
      Display *display;
      XID deviceid;
      int *modmap;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the device whose whose keymap is to be changed.

modmap Specifies a pointer to an XModifierKeymap structure.

`XHPSetDeviceModifierMapping` allows a client program to define the keycodes that are to be used as modifiers for an extended input device (other than the X keyboard and X pointer devices). The specified device must have previously been opened (turned on) with `XHPSetInputDevice`.

`XHPSetDeviceModifierMapping` specifies the KeyCodes of the keys, if any, that are to be used as modifiers for the specified input device. X permits up to eight modifier keys. If more than eight are specified in the `XModifierKeymap` structure, a `BadLength` error is generated.

There are eight modifiers, and 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 nonzero KeyCodes have meaning in each set (zero KeyCodes are ignored). If a nonzero KeyCode is given outside the range specified by min_keycode and max_keycode in the Display structure, or a KeyCode appears more than once in the entire map, a BadValue error is generated.

An X server can impose restrictions on how modifiers can be changed (for example, if certain keys do not generate up transitions in hardware 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, the status reply is MappingBusy, and no modifier is changed. XHPSetDeviceModifierMapping generates a DeviceMappingNotify event when it returns MappingSuccess.

XHPSetDeviceModifierMapping can generate BadDevice, BadLength, and BadValue errors.

E.1.26 Getting the Modifier Mapping of Extended Input Devices

To get the modifier mapping of an extended input device, use XHPGetDeviceModifierMapping.

```
XHPGetDeviceModifierMapping(display, deviceid)
      Display *display;
      XID deviceid;
```

display Specifies the connection to the X server.

deviceid Specifies the ID of the device whose modifier map is requested.

XHPGetDeviceModifierMapping allows a client program to read and use the keys being used as modifiers for an extended input device.

XHPGetDeviceModifierMapping returns a newly created XModifierKeymap structure that contains the keys being used as modifiers for the specified device. 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.

XHPGetDeviceModifierMapping can generate a BadDevice error.

E.1.27 Getting the Server Mode

Some displays have both image and overlay planes. For those displays, there are four combinations of image and overlay planes in which the server can run. To get the current mode of a specified screen, use XHPGetServerMode.

```
XHPGetServerMode(display, screen)
      Display *display;
      int screen;
```

display Specifies the connection to the X server.

screen Specifies the number of the screen whose mode is requested.

XHPGetServerMode allows a client program to determine the mode of a particular screen. The mode returned is an integer that can be compared against the following predefined modes:

XHPOVERLAY_MODE	The X server is running in the overlay planes.
XHPIMAGE_MDOE	The X server is running in the image planes.
XHPSTACKED_SCREENS_MODE	The X server is running with the overlay and image planes on different screens.
XHPCOMBINED_MODE	The X server is running in both the overlay and image planes.

These constants can be obtained by including the file <X11/XHPproto.h>. For more information on using these modes, refer to chapters 7 and 9 in *Using the X Window System* (HP part number 98794-90001).

If an invalid screen number is used, a -1 is returned by this function.

E.2 Image Input/Output Library Functions

The image I/O library functions describe in this section are provided to enable developers to produce window or pixmap hardcopy from within their application programs. These functions provide a path to and from image files stored in the *xwd* format.

The functions all return a zero result on successful completion. Integer error numbers (defined in <X11/XHPImageIO.h>) are returned if problems are encountered.

E.2.1 Saving the Contents of a Window

To save the contents of a rectangular window area in a file, use `XHPWindowToFile`.

```
int XHPWindowToFile(display, w, x, y, width, height, plane_mask, format, filename)
    Display *display;
    Window w;
    int x, y;
    unsigned int width, height;
    long plane_mask;
    int format;
    char *filename;
```

display Specifies the connection to the X server.

w Specifies the window ID. This is the where the image to be saved is found.

x
y Specify the x and y coordinates. These coordinates define the upper left corner of the rectangle and are relative to the origin of the drawable.

width
height Specify the width and height of the subimage. These arguments define the dimensions of the rectangle.

plane_mask Specifies the plane mask.

format Specifies the format for the image. You can pass one of these constants: `XYPixmap` or `ZPixmap`.

filename Specifies the file name to use. The format of the file name is operating system specific.

The `XHPWindowToFile` function saves the specified window rectangle in the format defined by the `xwd` (*X Window Dump*) utility program. This stores a file header and color map along with the image.

The *plane_mask* parameter controls which image planes will be included in the file. A value of ~0 (or -1) can be given to have all image planes stored.

Images saved using `XHPWindowToFile` may be viewed using the `xwud` utility or restored under program control using `XHPFileToWindow` or `XHPFileToPixmap`.

Hardcopy of a saved image can be generated using the `xpr` utility or by translating the image into Starbase format using `xwd2sb` and piping the result to the `pcltrans` utility. This can be done under program control using the *system*(*3S*) library routine to issue the appropriate shell command.

E.2.2 Saving a Pixmap

To save the contents of a rectangular pixmap area in a file, use `XHPPixmapToFile`.

```
int XHPPixmapToFile(display, pixmap, color_w, x, y, width, height, plane_mask, format, filename)
    Display *display;
    Pixmap pixmap;
    Window color_w;
    int x, y;
    unsigned int width, height;
    long plane_mask;
    int format;
    char *filename;
```

display	Specifies the connection to the X server.
pixmap	Specifies the pixmap ID. This is the where the image to be saved is found.
color_w	Specifies a window ID. This window's colormap will be saved in the image file. Visual attributes associated with this window are used in constructing the image file header.
x *y*	Specify the x and y coordinates. These coordinates define the upper left corner of the rectangle and are relative to the origin of the drawable.
width *height*	Specify the width and height of the subimage. These arguments define the dimensions of the rectangle.
plane_mask	Specifies the plane mask.
format	Specifies the format for the image. You can pass one of these constants: XYPixmap or ZPixmap.
filename	Specifies the file name to use. The format of the file name is operating system specific.

The `XHPPixmapToFile` function is similar to `XHPWindowToFile` but requires an additional parameter to specify the color map to be stored with the image. If the *color_w* parameter is zero then the root window associated with the pixmap is used to derive visual attributes and the colormap which gets stored in the image file.

E.2.3 Displaying a Stored Image

To transfer an image stored in a file into a window, use `XHPFileToWindow`.

```
int XHPFileToWindow(display, w, modify_cmap, gc, src_x, src_y, dst_x, dst_y, width, height, filename)
    Display *display;
    Window w;
    int modify_cmap;
    GC gc;
    int src_x, src_y;
    int dst_x, dst_y;
    unsigned int width, height;
    char *filename;
```

display	Specifies the connection to the X server.
w	Specifies the window ID. This is where the image will be placed.
modify_cmap	Specifies color map modification. If zero the window's color map is unchanged, if nonzero the window's color map will be updated from color map data contained in the image file.
gc	Specifies the graphics context.
src_x *src_y*	Specify the x and y coordinates of the upper left corner of the rectangle to be transferred from the image file.
dst_x *dst_y*	Specify the x and y coordinates within the window where the upper left corner of the image will be drawn.
width *height*	Specify the width and height of the subimage. These arguments define the dimensions of the rectangle.
filename	Specifies the file name to use. The format of the file name is operating system specific.

The `XHPFileToWindow` function transfers an image saved in a file in the `xwd` (*X Window Dump*) format into a window.

The graphics context specified by the *gc* parameter is used to control image transfer details. Refer to the "Transferring Images Between Client and Server" section in chapter 6 of this manual.

If the *gc* parameter is zero then the default graphics context for the *display*'s default screen will be used.

E.2.4 Displaying a Stored Pixmap

To transfer an image stored in a file into a pixmap, use `XHPFileToPixmap`.

```
int XHPFileToPixmap(display, pixmap, cmap, gc, src_x, src_y, dst_x, dst_y, width, height, filename)
    Display *display;
    Pixmap pixmap;
    Colormap cmap;
    GC gc;
    int src_x, src_y;
    int dst_x, dst_y;
    unsigned int width, height;
    char *filename;
```

display Specifies the connection to the X server.

pixmap Specifies the pixmap ID. This is the where the image will be placed.

cmap Specifies the color map ID. If nonzero, this color map will be updated from the color map data contained in the image file.

gc Specifies the graphics context.

src_x
src_y Specify the x and y coordinates of the upper left corner of the rectangle to be transferred from the image file.

dst_x
dst_y Specify the x and y coordinates within the window where the upper left corner of the image will be drawn.

width
height Specify the width and height of the subimage. These arguments define the dimensions of the rectangle.

filename Specifies the file name to use. The format of the file name is operating system specific.

The `XHPFileToPixmap` function is similar to `XHPFileToWindow` but has a *cmap* parameter to directly specify the color map to be modified by the colormap stored in the image file. If *cmap* is zero no colormap modification will occur.

E.2.5 Getting the Image File Header Structure

Use `XHPQueryImageFile` to get an image file header structure for a particular image file. For example, you might use this function to determine the size (or other attributes) of an image before displaying it.

```
int XHPQueryImageFile(filename, xwd_header_return)
    char *filename;
    XWDFileHeader *xwd_header_return;
```

filename	Specifies the file name to use. The format of the file name is operating system specific.
xwd_header_return	Returns information about the stored image in the XWDFileHeader structure.

The file <X11/XWDFile.h> is listed here for reference. Using the `XHPQueryImageFile` function, the programmer can access information in an image file's header structure.

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

/* Copyright 1985, 1986, Massachusetts Institute of Technology */

/* $Header: XWDFile.h,v 1.1 87/09/23 10:05:36 leichner Exp $ */
/*
 * XWDFile.hMIT Project Athena, X Window system window raster
 *          image dumper, dump file format header file.
 *
 *  Author: Tony Della Fera, DEC
 *          27-Jun-85
 *
 * Modifier:    William F. Wyatt, SAO
 *              18-Nov-86  - version 6 for saving/restoring color maps
 */

typedef unsigned long xwdval;

#define XWD_FILE_VERSION 7

typedef struct _xwd_file_header {
      xwdval header_size;        /* Size of the entire file header (bytes). */
      xwdval file_version;       /* XWD_FILE_VERSION */
      xwdval pixmap_format;      /* Pixmap format */
      xwdval pixmap_depth;       /* Pixmap depth */
      xwdval pixmap_width;       /* Pixmap width */
      xwdval pixmap_height;      /* Pixmap height */
      xwdval xoffset;            /* Bitmap x offset */
      xwdval byte_order;         /* MSBFirst, LSBFirst */
      xwdval bitmap_unit;        /* Bitmap unit */
      xwdval bitmap_bit_order;   /* MSBFirst, LSBFirst */
      xwdval bitmap_pad;         /* Bitmap scanline pad */
      xwdval bits_per_pixel;     /* Bits per pixel */
      xwdval bytes_per_line;     /* Bytes per scanline */
      xwdval visual_class;       /* Class of colormap */
      xwdval red_mask;           /* Z red mask */
      xwdval green_mask;         /* Z green mask */
      xwdval blue_mask;          /* Z blue mask */
      xwdval bits_per_rgb;       /* Log base 2 of distinct color values */
      xwdval colormap_entries;   /* Number of entries in colormap */
      xwdval ncolors;            /* Number of Color structures */
      xwdval window_width;       /* Window width */
      xwdval window_height;      /* Window height */
      long window_x;             /* Window upper left X coordinate */
      long window_y;             /* Window upper left Y coordinate */
      xwdval window_bdrwidth;    /* Window border width */
} XWDFileHeader;
```

E.3 National Language I/O Support

The X Library (Xlib) supports input and output of both 8-bit and 16-bit characters in many situations. The 16-bit I/O capability is implemented by the National Language I/O subsystem available for HP 9000 computers. (The national language subsystem is available in several Asian languages.) This extends the standard X font functionality to provide

- mixed 8- and 16-bit character output for applications using the X11 Library.
- 16-bit character input and output for applications using the Xr11 library for input and output. See the *Programming with the Xrlib User Interface Toolbox* manual for more information.

National language I/O is supported for 16-bit character fonts that are indexed by "HP-15" code. Each font typically includes both 8-bit and 16-bit characters.

E.3.1 Xlib Support

The X11 Library (Xlib) provides transparent text handling capability, independent of the difference between 8-bit and 16-bit characters, for the following six Xlib functions.

- `XTextWidth`
- `XTextExtents`
- `XQueryTextExtents`
- `XDrawText`
- `XDrawString`
- `XDrawImageString`

For the these functions to use a single 8-bit and 16-bit mixed font, the following five Xlib functions provide the capability which concurrently loads and unloads separated 8-bit (font) and 16-bit (associate font) files.

- `XLoadFont`
- `XQueryFont`
- `XLoadQueryFont`
- `XFreeFont`
- `XUnloadFont`

If the following conditions are fulfilled when loading a font with XLoadFont, and XLoadQueryFont, an 8- and 16-bit mixed font will be loaded by Xlib, until XFreeFont or XUnloadFont are called.

1. There exists a language designation in the specified font.
2. The XLoadFont and XLoadQueryFont functions look for the language designation in the following order.

- First examine the value of the font property LANGUAGE. This is a 8-bit STRING type property.
- Next examine the value of the environment variable LANG.

  Currently, "japanese", "korean", "chinese-s", and "chinese-t" are supported as valid LANGUAGE property or LANG environment variable designations.
- There exists the associate font designation in the specified font.

  XLoadFont and XLoadQueryFont look for the associate font via the following mechanism:

- First examine the value of the font property ASSOCIATE_FONT. This is an 8-bit STRING type property.
- Next examine the value of the environment variable XASSOCFONT.
- If neither the ASSOCIATE_FONT property or XASSOCFONT environment variable are set, then the name of the font file is used as the associate font.

XLoadFont and XLoadQueryFont look for the font properties LANGUAGE and ASSOCIATE_FONT in the specified font first. If either or both are undefined, then the environment variables LANG and XASSOCFONT are examined instead. If neither properties or environment values are defined the name of the font file is used as the associate font designation.

If the logically mixed font is implicitly specified as the font argument for XTextWidth, XTextExtents, XQueryTextExtents, XDrawText, XDrawString, or XDrawImageString, then the string argument for these functions may point to a string containing mixed 8- and 16-bit characters encoded by HP-15. Otherwise, all the characters will be interpreted as 8-bit characters. This provides transparency with standard X11 fonts.

E.3.2 Getting the Associate Font

For a font, which includes both the language and the associate font designations, `XQueryFont` and `XLoadQueryFont` return a pointer to the `XFontStruct` structure of the specified font as expected. To obtain the `XFontStruct` of the associate font, use the `XHPGet16bitMixedFontStruct`.

```
XFontStruct *XHPGet16bitMixedFont(font)
        XFontStruct font;
```

font Specifies the font ID.

`XHPGet16bitMixedFontStruct` returns a pointer to an `XFontStruct` structure of the associated font, if the specified font is a mixed 8- and 16-bit font. If the font specified is not a 8- and 16-bit mixed font, then NULL is returned.

E.3.3 Checking for 16-bit Characters

To determine if two bytes are defined as a 16-bit character for a specified font, use `XHPIs16bitCharacter`.

```
Bool XHPIs16bitCharacter(font, byte1, byte2)
        Font font;
        unsigned char byte1,
                      byte2;
```

font specifies the font to check for a 16-bit character.

byte1 specifies the first byte of a 16-bit character.

byte2 specifies the second byte of a 16-bit character. `XHPIs16bitCharacter` returns `True` if *byte1* and *byte2* are defined as the first and second bytes of a 16-bit character. In this function, the 16-bit character is based on HP-15 encoding determined by the language designation included in the specified font.

E.3.4 Conversions Between X11 Keysyms and HP Roman 8 codes

To convert an X11 Keysym into an HP Roman 8 character, use the `XHPKeysymToRoman8` function.

```
int XHPKeysymToRoman8(keysym, r8_return)
       Keysym keysym;
       char *r8_return;     /* RETURN */
```

keysym Specifies an X11 KeySym.

r8_return Specifies a pointer to a location to receive the converted Roman 8 character to *keysym*, if any.

`XHPKeysymToRoman8` takes an X11 KeySym and converts it to an HP Roman 8 character. The character is returned to the location pointed to by *r8_return*. If no Roman 8 character for *keysym* exists, then `XHPKeysymToRoman8` returns 0 (zero) and **r8_return* remains unchanged.

Some Keysyms are unique to Hewlett-Packard equipment because Roman 8 contains characters that were not encoded in the Keysyms distributed by MIT. To convert an HP Roman 8 character into an X11 KeySym, use `XHPRoman8ToKeysym`.

```
Keysym XHPRoman8ToKeysym(r8_char)
       char r8_char;
```

`XHPRoman8ToKeysym` takes an HP Roman 8 character and returns a KeySym.

NOTE

Most of the KeySyms returned by `XHPRoman8ToKeysym` will be ISO Latin-1 and various terminal functions. Two of the characters in the Roman 8 set ('S' with caron and 's' with caron) convert to Keysyms in the ISO Latin-2 set.

E.4 Locking an X Display

To provide better security for workstations and allow client programs to disable the key sequence used to reset the X server, the following functions may be used.

E.4.1 Disabling the Reset Key Sequence.

The X server may be terminated by pressing a particular set of keys. By default, that set is left shift, control, and reset.

To disable the reset key sequence, use `XHPDisableReset`.

```
XHPDisableReset(display)
        Display display;
```

display specifies the display.

This function is intended for use by client programs such as xsecure that provide security to systems running the X Window System. If a client program disables the reset sequence and exits without reenabling it, the reset sequence is automatically enabled by the server.

`XHPDisableReset` will fail with a `BadAccess` error, if another client has already disabled the reset key sequence.

E.4.2 Enabling the Reset Key Sequence.

To enable the reset key sequence, use `XHPEnableReset`.

```
XHPEnableReset(display)
        Display display;
```

display specifies the display.

`XHPEnableReset` enables the key sequence that is pressed to reset the X server. This function will fail with a `BadAccess` error, if this client did not previously disable the key sequence with `XHPDisableReset`.

E.5 Support for Multiple Error Handlers

To establish multiple error handling routines for a single process (up to one routine per connection to the server), use `XHPSetErrorHandler`.

```
#include <X11/XHPlib.h>
typedef int (*PFI) ();
XHPSetErrorHandler(display, routine)
        Display         *display;
        int             (*routine) ();

int routine(display, error)
        Display         *display;
        XErrorEvent     *error;
```

This function registers with Xlib the address of a routine to handle X errors. It is intended to be used by libraries and drivers that wish to establish an error handing routine without interfering with any error handling routine that may have been established by the client program.

`XHPSetErrorHandler` records one error handling routine per connection to the server. Therefore, for a library or driver to set up its own error handling routine without affecting that of the client, the library or driver must first have established its own connection to the server via `XOpenDisplay`.

When an XErrorEvent is received by the client, which error handling routine is invoked is determined by the display associated with the error. If the display matches that associated with a driver error handling routine, that error handling routine is invoked. If it does not match any driver routine, the error handling routine established by the client, if any exists, is invoked. Otherwise, the default Xlib error handler is invoked.

XHPSetErrorHandler returns the address of the previously established error handler. If that error handler was the default error handler, NULL is returned.

A driver or library may remove its error handler by invoking XHPSetErrorHandler with a NULL error handling routine.

HP Window Manager Programmatic Interface

F

This appendix describes the programmatic interface to the Hewlett-Packard Window Manager (*hpwm*). The conventions presented here (and earlier in this manual) describe how clients can be written to be "good citizens" in the X environment.

The purpose of the programmatic interface is to allow clients to communicate preferences to the window manager. This includes information about the size and placement of the window on the screen, the name of the window, the image on the icon, and so on. The general X window management philosophy is that clients should work without knowing or caring which window manager is being used (or even whether one is being used at all). If a window manager is present, the client should abide by the decisions of that window manager. For example, if the window manager denies a resize request, the client should make do with its current size.

F.1 Window Management Calls

Clients communicate with the window manager through properties associated with top-level windows, synthetic events (generated using `XSendEvent()`) and standard X events. Programmatically this communication involves Xlib calls, either directly or through libraries such as the Xt Intrinsics. Clients may programmatically interact with hpwm (or any X window manager) in the following ways:

- **Implicit programmatic access.** In this case clients do not set up any window properties or execute any call that directly communicates with the window manager. Communication occurs when the state of the client window is changed (such as when the window is mapped, unmapped, configured, or has a colormap change). To work with hpwm, clients are not required to do anything more than what is required when a window manager is not being used.
- **High-level programmatic access.** To establish and maintain standard communications with hpwm, clients can make high-level Xlib calls (such as `XSetStandardProperties()`) or calls to certain libraries built on Xlib (such as the Xt Intrinsics calls `XtInitialize()` and `XtMainLoop()`). Developers are encouraged to use the Xt Intrinsics for client/window manager communication unless the client has some specialized window management requirements.

- **Low-level programmatic access.** Clients with special window management requirements can use low-level Xlib calls (such as `XStoreName()` and `XSetWMHints()`) to communicate with the window manager.

The following Xlib calls are typically used to communicate with hpwm:

- `XSetStandardProperties()` sets WM_NAME, WM_ICON_NAME, WM_HINTS, WM_COMMAND and WM_NORMAL_HINTS. It does not set WM_CLASS (which should be set to allow hpwm to be optimally configured for a particular class of client windows).
- `XStoreName()` sets the WM_NAME property (used for window titles).
- `XSetIconName()` sets the WM_ICON_NAME property (used for the icon label).
- `XSetCommand()` sets the WM_COMMAND property.
- `XSetWMHints()` sets the WM_HINTS property.
- `XSetNormalHints()` sets the WM_NORMAL_HINTS property.
- `XGetIconSizes()` gets a list of hpwm supported icon sizes.
- `XSetClassHint()` sets the WM_CLASS property.
- `XSetTransientForHint()` sets the WM_TRANSIENT_FOR property.
- `XGetStandardColormap()` gets standard colormap information.

The following Xt Intrinsics calls are typically used to communicate with hpwm (refer to the *Programming With the Xt Intrinsics* manual for a complete description of each function):

- `XtInitialize()` makes a top-level window and sets up the WM_NAME, WM_ICON_NAME, WM_NORMAL_HINTS, WM_HINTS, WM_COMMAND and WM_CLASS properties.
- `XtCreateApplicationShell()` creates a top-level window and sets up the WM_NAME, WM_ICON_NAME, WM_NORMAL_HINTS, WM_HINTS, WM_CLASS, WM_COMMAND, and WM_TRANSIENT_FOR (for transient shell class widgets) properties.
- `XtMainLoop()` handles window reconfiguration messages.

F.2 Creating a Top-Level Window

When a window is created with `XCreateSimpleWindow()`, client properties must be established using calls such as `XStoreName()`. The recommended alternative to using `XCreateSimpleWindow()` is to use the Xt Intrinsics to create a top-level window.

F.2.1 Client Properties

This section supplements the information provided in chapter 9, "Predefined Property Functions."

WM_NAME

The WM_NAME string is displayed in the title area of the client window frame. The HP Window Manager dynamically changes the window title if the WM_NAME property value is changed by the client.

If this property is not set, the res_name part of the WM_CLASS property is used as the window title. If res_name is undefined, "*****" is used as the window title.

It is assumed that the encoding of the string passed in the WM_NAME property is compatible with the font being used for the window title.

WM_ICON_NAME

The WM_ICON_NAME string is displayed in the label part of the client's icon. The HP Window Manager dynamically changes the displayed icon title if the WM_ICON_NAME property value is changed by the client.

If this property is not set, the icon name is set using the window title.

It is assumed that the encoding of the string passed in the WM_ICON_NAME property is compatible with the font being used for the icon label.

WM_NORMAL_HINTS

The fields of the WM_NORMAL_HINTS property are *flags*, *min_width*, *min_height*, *max_width*, *max_height*, *width_inc*, *height_inc*, *min_aspect*, and *max_aspect*.

flags:

If the window size and position are specified by the user (using `USPosition` or `USSize`), hpwm places the window on the screen based on the configured window position and size. If the window position is not provided by the user *and* hpwm is configured for interactive placement, the user is allowed to interactively position or size the window on the screen. Otherwise, the configured window position and size are used. Initial window placement is affected by the hpwm `positionIsFrame` and `positionOnScreen` resource settings.

min_width, min_height:

If min_width or min_height is not greater than 0 or has not been set, a value of 1x1 or larger is used by hpwm. The actual minimum size used by hpwm is based on the minimum frame size for the frame type being used.

max_width, max_height:

If the `maximumClientSize` resource is not specified, max_width and max_height are used to set a maximum client window size. If max_width or max_height is not set, the maximum window size is set such that when the window is at its maximum size the window and window frame exactly fit the screen. If (max_width / max_height) is less than (min_width / min_height), the maximum window size is set to (min_width / min_height). The maximum size is limited if the `maximumMaximum` resource is specified. The HP Window Manager maximize function makes the window the maximum size.

width_inc, height_inc:

When sizing windows, hpwm reports the current window size in a status window. The units of size are in terms of the width_inc and height_inc. If width_inc and height_inc are not set, the sizing increment is set to 1 pixel.

min_aspect, max_aspect:

The HP Window Manager does not apply the aspect ratio constraint.

Changes to the WM_NORMAL_HINTS property are tracked by the window manager. Changes to the size and position fields are ignored, and changes to other fields affect subsequent window reconfiguration.

WM_HINTS

The fields of the WM_HINTS property are *flags*, *input*, *initial_state*, *icon_pixmap*, *icon_window*, *icon_x*, *icon_y*, *icon_mask*, *window_group*.

Except for changes to the icon_pixmap, the WM_HINTS property is only interpreted by hpwm when the client window goes from the *withdrawn* state (that is, when the window is not managed by hpwm) to the *normal* or *iconic* state.

flags:

This field identifies which of the fields are defined.

input:

This field is ignored by hpwm. If the user selects a window to have the keyboard input focus, that window is given the focus event even if this field is set to 0 (*false*). The client can always ignore keyboard input.

initial_state:

The value of this field determines the initial state of the client when its top-level window is mapped. A value of 1 causes the window to be visible (NormalState); a value of 3 causes the icon to be visible (IconicState).

icon_pixmap:

If the icon_pixmap is larger than the maximum icon image size (set by the hpwm `iconImageMaximum` resource), it is clipped to the maximum size. If the icon_pixmap is smaller than the minimum icon image size (set by the hpwm `iconImageMinimum` resource), it is not used. If the icon_pixmap is being used for the icon image (that is, an icon_window is not specified and the user has not specified an icon for this class of client window), hpwm changes the icon image when the icon_pixmap is changed.

The foreground and background colors for the icon_pixmap are specified in the hpwm resource files. (Many other resources may also be specified. Refer to *Using the X Window System*, HP part number 98794-90001.)

icon_x, *icon_y*:

The (icon_x, icon_y) coordinate is a hint to hpwm for the icon position.

icon_mask:

The icon_mask value is not used by hpwm.

icon_window:

Icon windows are supported by hpwm. If the icon_window is larger than the maximum icon image size (set by the `iconImageMaximum` resource), it is reconfigured to the maximum size. If the icon_window is smaller than the minimum icon image size (set by the `iconImageMinimum` resource), it is reconfigured to the minimum size. If both the icon_window and icon_pixmap are passed, the icon_window is used for the icon image.

window_group:

The window_group value is not used by hpwm.

WM_PROTOCOLS

The WM_PROTOCOLS property is a list of atoms. Each atom identifies a protocol in which the client is willing to participate. Atoms can identify both standard protocols and private protocols specific to individual window managers. At present, there are three standard protocols:

WM_SAVE_YOURSELF:

Clients including this atom will be notified when a session manager or a window manager wishes the window's state to be changed, typically because the window is about to be deleted, or the session terminated.

WM_TAKE_FOCUS:

Clients including this atom will be notified when a window manager believes that the client should explicitly set the input focus to one of its windows.

WM_DELETE_WINDOW:

Clients are notified when the hpwm *f.kill* function is invoked by the user. The HP Window Manager does not terminate the client or destroy the window when a WM_DELETE_WINDOW notification is done.

A client message event (the event type is ClientMessage) is used for WM_PROTOCOLS client notification. The client message has the following characteristics:

- The type is WM_PROTOCOLS.
- The format is 32.
- The atom naming the protocol (such as WM_DELETE_WINDOW) is in the data[0] field.
- A time stamp is in the data[1] field.

WM_CLASS

The fields of the WM_CLASS property are *res_class* and *res_name*.

res_class:

The res_class value is used by hpwm to configure window decorations and icons for windows associated with a particular client class. If the WM_CLASS property is not set, no special client class customization is done.

res_name:

The res_name value is only used by hpwm when the WM_NAME property is not set. In that case, the res_name value is used for the window title.

The WM_CLASS property is only interpreted by hpwm when the client window goes from the *withdrawn* state to the *normal* or *iconic* state.

WM_TRANSIENT_FOR

Transient windows are placed on the screen without user interaction. The window size and position information is used even if it was generated by the client program and not the user. Transient windows generally get less decoration than normal top level windows; this is controlled by the hpwm `transientDecoration` resource. When the normal client window associated with a transient window is minimized, the transient window is removed from the screen (unmapped). When the associated client window is normalized, the transient window is placed on the screen (mapped).

WM_COLORMAP_WINDOWS

This property is used to indicate to the window manager which colormaps a client would like to have installed. It is a property of the WINDOW that is a list of the IDs of windows that may need colormaps installed. That is, these colormaps differ from the colormap of the top-level client window.

If the WM_COLORMAP_WINDOWS property is present when the client window goes from the *withdrawn* state to the *normal* or *iconic* state, hpwm compiles a list of colormaps using the colormap attribute of the windows identified in the property along with the colormap attribute of the top-level client window. The HP Window Manager installs the colormaps subject to the colormap focus policy that has been selected by the user. The HP Window Manager monitors the colormap windows for colormap attribute changes and updates its colormap list accordingly. If the WM_COLORMAP_WINDOWS property is not present, hpwm installs the colormap indicated by the colormap attribute of the top-level client window.

F.3 Window Manager Properties

The HP Window Manager uses properties to supply configuration and presentation state information to clients.

WM_ICON_SIZE

The HP Window Manager sets the WM_ICON_SIZE property on the root window. This property contains information corresponding to an XIconSize structure (refer to section 9.1.7,"Setting and Getting Icon Size Hints"). The items in the XIconSize structure are *min_width*, *min_height*, *max_width*, *max_height*, *width_inc*, *height_inc*.

min_width, *min_height*:

min_width and min_height are set based on the value of (or default for) the `iconImageMinimum` resource.

max_width, *max_height*:

max_width and max_height are set based on the value of (or default for) the `iconImageMaximum` resource.

width_inc, *height_inc*:

The HP Window Manager sets width_inc and height_inc to 1.

F.4 Client Responses to Window Manager Actions

This section describes client responses to hpwm actions.

F.4.1 Redirection of Operations

The HP Window Manager redirects the following client top-level window requests: MapWindow, ConfigureWindow, CirculateWindow. Clients must not rely on immediate execution of redirected requests.

F.4.2 Window Configuration

Clients can hint to hpwm desirable window positions, but they must be able to accept the window positions that they are given.

Clients can hint to hpwm desirable window sizes, but they must be able to accept the window sizes that they are given. If a client cannot be useful in the window size that is given, it could display a message asking the user to resize the window.

Clients receive ConfigureNotify events in response to configuration requests as long as there is not an X error. This is true even if the window configuration was not changed.

Window coordinates in the ConfigureNotify event may be relative to the hpwm client frame window. *Clients must use XTranslateCoordinates to get root window relative coordinates.*

F.4.3 (De)Iconify

The HP Window Manager maps the client window when the window is to be displayed in its normal state and unmaps the client window when it is to be displayed in its iconic state. Client-supplied icon windows are mapped when the associated client window is in the iconic state, otherwise they remain unmapped.

F.4.4 Colormap Change

Clients that wish to be notified when their colormaps are installed or uninstalled should select ColormapNotify on client windows that have unique colormaps.

F.4.5 InputFocus

Clients should generally avoid the use of XSetInputFocus (even if one of their top-level windows has the input focus). The Xt Intrinsics and the HP X Widgets can be used to handle the distribution of input within a client window.

F.4.6 ClientMessage Events

Although there is no way for clients to prevent themselves being sent ClientMessage events, these events can be safely ignored if they are not useful. The HP Window Manager does not require clients to handle any ClientMessage events.

Example Programs G

This appendix contains the following example programs:

- `simple.c`, which creates a simple window and displays a static text message in it.
- `input.c`, which demonstrates how to get input from an extended input device.
- `depth.c`, which demonstrates how to create a window with a visual type different than its parent.

G.1 A Simple Example

Here's a simple program that creates a window and displays the static text string "Text inside the simple window." in it. By editing the definitions at the beginning of the program, you can change the window's name or icon name, the string that is displayed, and the font used.

```
/************************************************************************
 *
 *  File:     simple.c
 *
 *  This program creates a window and displays text in it.
 *  It uses the Xlib facilities, and does not support the X database
 *  mechanism to allow the user to override hard-coded defaults.
 *
 ************************************************************************/

#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define NAME "A Simple Window"
#define ICON_NAME "Simple"
#define STRING "Text inside the simple window."
#define FONT "vbee-36"

/*
 * Define the window manager hints.
 */
```

```
XWMHints xwmh = {
   (InputHint|StateHint), /* flags */
   False,         /* input -- ignored by hpwm */
   NormalState,  /* initial_state */
   0,             /* icon pixmap */
   0,             /* icon window */
   0, 0, /* icon location */
   0,             /* icon mask */
   0,             /* window group -- ignored by hpwm */
};

main (argc, argv)
int   argc;
char *argv[];
{
   unsigned     fontheight, pad, fg, bg, bd, bw;
   Display     *dpy;
   Window       win;
   GC           gc;
   XFontStruct *fontstruct;
   XEvent       event;
   XSizeHints   xsh;
   XWindowAttributes     xwa;
   XSetWindowAttributes  xswa;

   /*
    * Open the display using the DISPLAY environment variable to locate
    * the X server.
    */

   if ((dpy = XOpenDisplay(NULL)) == NULL) {
      fprintf (stderr,
               "%s: can't open %s.\n", argv[0], XDisplayName(NULL));
      exit(1);
   }

   /*
    * Load the font to use.
    */

   if ((fontstruct = XLoadQueryFont(dpy, FONT)) == NULL) {
      fprintf (stderr,
               "%s: display %s doesn't know font %s.\n",
               argv[0], DisplayString(dpy), FONT);
      exit(1);
   }
   fontheight = fontstruct->max_bounds.ascent + fontstruct->max_bounds.descent;

   /*
    * Select colors for the border, the window background, and the
    * window foreground.
    */

   bd = WhitePixel(dpy, DefaultScreen(dpy));
   bg = BlackPixel(dpy, DefaultScreen(dpy));
   fg = WhitePixel(dpy, DefaultScreen(dpy));
```

```
/*
 * Set the border width and padding.
 */

bw = 1;
pad = 1;

/*
 * Fill out the XSizeHints structure for initial window position
 * and size.
 */

xsh.flags = (PPosition|PSize);
xsh.height = fontheight + 2 * pad;
xsh.width = XTextWidth(fontstruct, STRING, strlen(STRING)) + 2 * pad;
xsh.x = (DisplayWidth(dpy, DefaultScreen(dpy)) - xsh.width) / 2;
xsh.y = (DisplayHeight(dpy, DefaultScreen(dpy)) - xsh.height) / 2;

/*
 * Create the unmapped window.
 */

win = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
       xsh.x, xsh.y, xsh.width, xsh.height, bw, bd, bg);

/*
 * Set the standard properties and window manager hints for the window.
 */

XSetStandardProperties(dpy, win, NAME, ICON_NAME, None, argv, argc,
   &xsh);
XSetWMHints(dpy, win, &xwmh);

/*
 * Ensure that the window's colormap field points to the default
 * colormap.  Set the window's Bit Gravity to reduce Expose events.
 */

xswa.colormap = DefaultColormap(dpy, DefaultScreen(dpy));
xswa.bit_gravity = CenterGravity;
XChangeWindowAttributes(dpy, win, (CWColormap|CWBitGravity), &xswa);

/*
 * Create the GC for writing text.
 */

gc = DefaultGC(dpy, DefaultScreen(dpy));
XSetFont(dpy, gc, fontstruct->fid);
XSetForeground(dpy, gc, fg);
XSetBackground(dpy, gc, bg);

/*
 * Specify the event types we are interested in - only exposures.
 */

XSelectInput(dpy, win, ExposureMask|StructureNotifyMask);
```

```
/*
 * Map the window.
 */

XMapWindow(dpy, win);

/*
 * Loop forever, examining each event.
 */

while (1) {

/*
 * Get the next event.
 */

   XNextEvent(dpy, &event);

/*
 * Repaint the window on the last Expose or ConfigureNotify event.
 */

   if ((event.type == ConfigureNotify) ||
       (event.type == Expose)) {
      int x, y;

/*
 * Find out how big the window is now.
 */

      if (XGetWindowAttributes(dpy, win, &xwa) == 0)
         break;
      x = (xwa.width - XTextWidth(fontstruct, STRING, strlen(STRING)))/2;
      y = (xwa.height + fontstruct->max_bounds.ascent
                      - fontstruct->max_bounds.descent)/2;

/*
 * Fill the window with the background color.
 * Paint the centered string.
 */

      XClearWindow(dpy, win);
      XDrawString(dpy, win, gc, x, y, STRING, strlen(STRING));

/*
 * Remove pending Expose events from the event queue to avoid
 * multiple repaints.
 */
      while (XCheckTypedEvent(dpy, Expose, &event));
   }
}

fprintf (stderr, "Can't get window attributes.\n");
exit(1);

}
```

G.2 Getting Input From an Extended Input Device

This program demonstrates how to get input from an extended input device (that is, a device other than the standard X keyboard or pointer).

`input.c` creates two windows, enables all input devices other than the X keyboard and X pointer devices, and selects input from them when the X pointer is in the smaller of the two windows.

When a button is pressed, or a valuator moved on one of those other devices, and the X pointer is in the created window, the contents of the events generated by the other devices are displayed.

```
/*************************************************************************
 *
 * File: input.c
 *
 * Sample program to enable all extension input devices and select all
 * input events from them.  This program creates 2 windows and selects
 * input from the smaller of the two.
 *
 * To terminate this program, press button 1 on some extension device
 * when the X pointer is in the window from which input has been selected.
 *
 * To compile this program, use: "cc input.c -lXhp11 -lX11 -o input"
 *
 */
#include <X11/Xlib.h>
#include <X11/XHPlib.h>
#include <X11/Xutil.h>
#include "stdio.h"


Display *display;
Window  root;
int     devicekeypress;
int     devicekeyrelease;
int     devicebuttonpress;
int     devicebuttonrelease;
int     devicemotionnotify;
int     devicefocusin;
int     devicefocusout;
int     proximityin;
int     proximityout;
int     devicekeymapnotify;
int     devicemappingnotify;
```

```
main ()
    {
    XHPDeviceList       *slist;
    int         ndevices;
    Window              my;
    Window              my2;
    XEvent              event;
    unsigned            int     mask;
    XHPDeviceList       *list;

    display = XOpenDisplay ("");
    if (display == NULL)
        {
        printf ("No connection to server - aborting example.\n");
        exit(1);
        }
    root = RootWindow (display,0);

    create_two_windows (&my, &my2);
    get_all_masks (&mask);
    ndevices = enable_all_devices (mask, &slist);
    select_ext_input (my2, slist, mask, ndevices);

    for (;;)
        {
        XNextEvent (display,&event);
        if (process_device_events (&event) == -1)
            break;
        }

    close_all_devices (slist, ndevices);
    XHPFreeDeviceList (slist);
    }

/***********************************************************************
 *
 * This function gets the event masks and event types for all extension events.
 *
 */

get_all_masks (mask)
    unsigned            int     *mask;
    {
    unsigned            int     tmask;
    unsigned            int     event;

    XHPGetExtEventMask (display, HPDeviceKeyPressreq, &devicekeypress, &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPDeviceKeyReleasereq, &devicekeyrelease,
        &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPDeviceButtonPressreq, &devicebuttonpress,
        &tmask);
    *mask |= tmask;
```

```
    XHPGetExtEventMask (display, HPDeviceButtonReleasereq,
        &devicebuttonrelease, &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPDeviceMotionNotifyreq, &devicemotionnotify,
        &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPDeviceFocusInreq, &devicefocusin, &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPDeviceFocusOutreq, &devicefocusout, &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPProximityInreq, &proximityin, &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPProximityOutreq, &proximityout, &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPDeviceKeymapNotifyreq, &devicekeymapnotify,
        &tmask);
    *mask |= tmask;

    XHPGetExtEventMask (display, HPDeviceMappingNotifyreq,
        &devicemappingnotify, &tmask);
    *mask |= tmask;
    }

/************************************************************************
 *
 * This function lists and enables all extension devices.
 *
 */

enable_all_devices (mask, slist)
    unsigned int mask;
    XHPDeviceList        **slist;
    {
    int                  ndevices;
    int                  ret, i;
    XHPDeviceList        *list;

    *slist = XHPListInputDevices (display, &ndevices);
    printf ("The number of available input devices is %d\n",ndevices);
    for (i=0,list=(*slist); i<ndevices; i++,list++)
        {
        if (list->x_id != XPOINTER && list->x_id != XKEYBOARD)
            {
            ret = XHPSetInputDevice (display, list->x_id, (ON | DEVICE_EVENTS));
            if (ret == 0)
                printf ("Enabled %s\n",list->name);
            }
        }
    printf("\n");
    return (ndevices);
    }
```

```
/************************************************************************
 *
 * This function selects for all extension events from all extension
 * devices.
 *
 */

select_ext_input (win, slist, mask, ndevices)
    Window win;
    XHPDeviceList        *slist;
    unsigned int mask;
    int ndevices;
    {
    int i;
    XHPDeviceList        *list;


    for (i=0, list=slist; i<ndevices; i++, list++)
        {
        if (list->x_id != XPOINTER && list->x_id != XKEYBOARD)
            XHPSelectExtensionEvent (display, win, list->x_id, mask);
        }
    }

/************************************************************************
 *
 * This function closes (turns off) all extension devices.
 *
 */

close_all_devices (slist, ndevices)
    XHPDeviceList        *slist;
    int                  ndevices;
    {
    int                  ret, i;
    XHPDeviceList        *list;

    for (i=0,list=slist; i<ndevices; i++,list++)
        {
        if (list->x_id != XPOINTER && list->x_id != XKEYBOARD)
            {
            ret = XHPSetInputDevice (display, list->x_id, (OFF));
            if (ret == 0)
                printf ("Disabled %s\n",list->name);
            }
        }
    printf("\n");
    return (ndevices);
    }

/************************************************************************
 *
 * This function creates two windows.  The smaller will be used to
 * select input from all extension devices.
 *
 */
```

```
create_two_windows (my, my2)
    Window *my, *my2;
    {
    XSetWindowAttributes attributes;
    unsigned long        attribute_mask;
    int          status;
    XSizeHints           hints;
    Screen               *screen = XDefaultScreenOfDisplay (display);

    attribute_mask = CWBackPixmap;
    attribute_mask = CWBackPixel;
    attribute_mask |= CWEventMask;
    attributes.background_pixmap = None;
    attributes.background_pixel = WhitePixel(display, 0);
    attributes.event_mask = ExposureMask;

    *my = XCreateWindow (display, root, 100,100, 400,200,1,
        DefaultDepthOfScreen (screen),
        InputOutput, CopyFromParent, attribute_mask, &attributes);

    if (*my == 0) {
        fprintf (stderr, "can't create window!\n");
        exit (1);
    }
    status = XGetNormalHints (display, *my, &hints);
    hints.flags |= (USPosition | USSize | PPosition | PSize);
    XSetNormalHints (display, *my, &hints);
    XMapWindow (display, *my);
    XFlush(display);

    attribute_mask = CWBackPixmap;
    attribute_mask = CWBackPixel;
    attribute_mask |= CWEventMask;
    attributes.background_pixmap = None;
    attributes.background_pixel = BlackPixel(display, 0);
    attributes.event_mask = ExposureMask;

    *my2 = XCreateWindow (display, *my, 50,50, 300,100,1,
        DefaultDepthOfScreen (screen),
        InputOutput, CopyFromParent, attribute_mask, &attributes);
    if (my2 == 0) {
        fprintf (stderr, "can't create window!\n");
        exit (1);
    }
    status = XGetNormalHints (display, *my2, &hints);
    hints.flags |= (USPosition | USSize | PPosition | PSize);
    XSetNormalHints (display, *my2, &hints);
    XMapWindow (display, *my2);
    XFlush(display);

    }

/*************************************************************************
 *
 * This function figures out what kind of device event we received.
 *
 */
```

```
process_device_events (event)
    XEvent       *event;
    {
    int                          i;
    XHPDeviceMotionEvent *m;
    XHPDeviceKeyEvent            *k;
    XHPDeviceButtonEvent *b;
    XHPProximityNotifyEvent      *p;
    XHPDeviceFocusChangeEvent   *f;
    XHPDeviceKeymapEvent         *n;
    XHPDeviceMappingEvent        *q;

    XExposeEvent         *e;
    XAnyEvent                    *x;

    if (event->type == devicekeypress)
        {
        k = (XHPDeviceKeyEvent * ) event;
        printf ("Device key press event device=%d\n", k->deviceid);
        printf ("     type =        %d\n", k->ev.type);
        printf ("     serial =      %ld\n", k->ev.serial);
        printf ("     send_event =  %ld\n", k->ev.send_event);
        printf ("     display =     %x\n", k->ev.display);
        printf ("     window =      %x\n", k->ev.window);
        printf ("     root =        %x\n", k->ev.root);
        printf ("     subwindow =   %x\n", k->ev.subwindow);
        printf ("     time =        %x\n", k->ev.time);
        printf ("     x =           %d\n", k->ev.x);
        printf ("     y =           %d\n", k->ev.y);
        printf ("     x_root =      %d\n", k->ev.x_root);
        printf ("     y_root =      %d\n", k->ev.y_root);
        printf ("     state =       %d\n", k->ev.state);
        printf ("     keycode =     %x\n", k->ev.keycode);
        printf ("     same_screen = %d\n", k->ev.same_screen);
        }

    else if (event->type == devicekeyrelease)
        {
        k = (XHPDeviceKeyEvent * ) event;
        printf ("Device key release event received from device %d\n",
            k->deviceid);
        }
```

```
else if (event->type == devicebuttonpress)
    {
    b = (XHPDeviceButtonEvent * ) event;
    printf ("Device button press event device=%d\n", b->deviceid);
    printf ("     type =         %d\n", b->ev.type);
    printf ("     serial =       %ld\n", b->ev.serial);
    printf ("     send_event =   %ld\n", b->ev.send_event);
    printf ("     display =      %x\n", b->ev.display);
    printf ("     window =       %x\n", b->ev.window);
    printf ("     root =         %x\n", b->ev.root);
    printf ("     subwindow =    %x\n", b->ev.subwindow);
    printf ("     time =         %x\n", b->ev.time);
    printf ("     x =            %d\n", b->ev.x);
    printf ("     y =            %d\n", b->ev.y);
    printf ("     x_root =       %d\n", b->ev.x_root);
    printf ("     y_root =       %d\n", b->ev.y_root);
    printf ("     state =        %d\n", b->ev.state);
    printf ("     button =       %x\n", b->ev.button);
    printf ("     same_screen =  %d\n", b->ev.same_screen);
    if (b->ev.button == 1)           /* this causes us to quit */
        return (-1);
    }

else if (event->type == devicebuttonrelease)
    {
    b = (XHPDeviceButtonEvent * ) event;
    printf ("Device button release event received from device %d\n",
        b->deviceid);
    }

else if (event->type == devicemotionnotify)
    {
    m = (XHPDeviceMotionEvent * ) event;
    printf ("DeviceMotionNotify event received from device=%d\n",
            m->deviceid);
    printf ("     type =         %d\n", m->ev.type);
    printf ("     serial =       %ld\n", m->ev.serial);
    printf ("     send_event =   %ld\n", m->ev.send_event);
    printf ("     display =      %x\n", m->ev.display);
    printf ("     window =       %x\n", m->ev.window);
    printf ("     root =         %x\n", m->ev.root);
    printf ("     subwindow =    %x\n", m->ev.subwindow);
    printf ("     time =         %x\n", m->ev.time);
    printf ("     x =            %d\n", m->ev.x);
    printf ("     y =            %d\n", m->ev.y);
    printf ("     x_root =       %d\n", m->ev.x_root);
    printf ("     y_root =       %d\n", m->ev.y_root);
    printf ("     state =        %d\n", m->ev.state);
    printf ("     is_hint =      %x\n", m->ev.is_hint);
    printf ("     same_screen =  %d\n", m->ev.same_screen);
    for (i=0; i<m->axes_count; i++)
        printf ("     motion data for axis %d is %d\n",
            m->data[i].ax_num, m->data[i].ax_val);
    }
```

```
    else if (event->type == proximityin)
        {
        p = (XHPProximityNotifyEvent * ) event;
        printf ("ProximityIn event received from device %d\n", p->deviceid);
        }

    else if (event->type == proximityout)
        {
        p = (XHPProximityNotifyEvent * ) event;
        printf ("ProximityOut event received from device=%d\n",
            p->deviceid);
        }

    else if (event->type == devicefocusin)
        {
        f = (XHPDeviceFocusChangeEvent * ) event;
        printf ("DeviceFocusIn event received from device %d\n",f->deviceid);
        }

    else if (event->type == devicefocusout)
        {
        f = (XHPDeviceFocusChangeEvent * ) event;
        printf ("DeviceFocusOut event received from  device %d\n",
            f->deviceid);
        }

    else if (event->type == devicekeymapnotify)
        {
        n = (XHPDeviceKeymapEvent * ) event;
        printf ("Device Keymap notify event received from device %d\n",
                n->deviceid);
        }

    else if (event->type == devicemappingnotify)
        {
        q = (XHPDeviceMappingEvent * ) event;
        printf ("Device Mapping notify event received from device %d.\n",
                q->deviceid);
        }

    else
        switch (event->type)
            {
            case Expose:
                e = (XExposeEvent * ) event;
                printf ("Exposure notify event received.\n");
                break;
            default:
                x = (XAnyEvent *) event;
                printf ("Got an event of type %d\n", x->type);
            }

}
```

G.3 Using Image and Overlay Planes

This program demonstrates the minimum necessary steps to create an X window whose visual type is different than that of its parent. This program is specifically tailored to look for a visual whose depth is 8 and whose class is `PseudoColor`. (The steps are the same for other values of depth and class.)

As long as the parent window's class and depth are different than the window being created, certain additional operations *must* be performed before the window can be created. In particular, there are two mandatory steps:

- A colormap must be created or obtained otherwise and given to the window at create time.
- A border pixel or pixmap must be created or otherwise obtained and given to the window at create time.

Other than these two requirements, everything else is the same as for creating any other window.

```
/***********************************************************************
 *
 * File:     depth.c
 *
 * This program creates a window and displays text in it.  This program
 * looks specifically for a visual whose depth is 8 and whose class is
 * PseudoColor.
 *
 ***********************************************************************/

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

#define DEPTH 8         /* Desired Depth */
#define WHITE 1
#define BLACK 0
#define BIG_STRING "ABCEFGHIJKLMNOPRSTUVWXYZ1234567890abcdefghijklmnopqrstuvwx
yz1234567890/.,<>?;:]"
#define WIDTH 80        /* Width in characters of the window */
#define HEIGHT 24       /* Height in characters of the window */
#define X_ORG    100     /* X Origin of the window on screen */
#define Y_ORG    100     /* y Origin of the window on screen */

char    FontName[128] = "hp8.8x16b";
```

```
char *colors[] =
{
        "black",
        "white",
        0
};

main()
{
        Display *dpy;
        XVisualInfo *pVisInfo,visInfo;
        int retVal;
        Colormap cmapID;
        XColor exactC,defC;
        Window w;
        XSetWindowAttributes wAttr;
        char **ppColor;

        char *display = NULL;
        int fg, bg;
        int i;
        int yPos;
        Font    myFont;
        XFontStruct      *myFontStruct;
        XEvent myEvent;

        Window win;
        int charHeight, charWidth;
        int winX, winY, winW, winH;
        unsigned int mask;
        XSetWindowAttributes xswa;

        GC gc;
        XGCValues xgcv;

        /*
         * The first step, of course, is to open the display
         */

        dpy = XOpenDisplay(0);
        if (!dpy)
        {
                fprintf(stderr,"Could't open display: %s\n",getenv("DISPLAY"));
                exit(1);
        }

        /*
         * Next we'll get the font that we will be using and get
         * some information from it which will be used to determine
         * window size.
         */
```

```
if(myFontStruct = XLoadQueryFont(dpy, &FontName[0]))
{
    myFont = myFontStruct->fid;
    charHeight = myFontStruct->max_bounds.ascent +
                        myFontStruct->max_bounds.descent;
    charWidth = myFontStruct->max_bounds.width;
}
else
{
    printf("Couldn't load font %s...Bye!\n", &FontName[0]);
    exit(1);
}

/*
 * Now we will ask the server for the visual type and depth
 * that we are interested in.
 */

visInfo.screen = 0;
visInfo.depth = DEPTH;
visInfo.class = PseudoColor;
mask = VisualScreenMask | VisualDepthMask | VisualClassMask;

pVisInfo = XGetVisualInfo(dpy, mask, &visInfo, &retVal);

if (!retVal)
{
        fprintf(stderr,"Could not get visual info\n");
        exit(1);
}
if (retVal != 1)
{
        fprintf(stderr,"Too many visuals match display+depth+class\n");
        exit(1);
}


/*
 * At this point, we have the visual information that we need.
 * In order to create a window, we have to create a colormap
 * for this visual class (assuming that it is different than
 * the default visual class.
 */

cmapID = XCreateColormap(dpy,
                         RootWindowOfScreen(ScreenOfDisplay(dpy,0)),
                         pVisInfo->visual, AllocNone);
if (!cmapID)
{
        fprintf(stderr,"Could not create color map\n");
        exit(1);
}
```

```
/*
 * Since this is a brand new colormap, we need to allocate
 * some colors in it.  The initial colormap may not be exactly
 * what we need.
 */

ppColor = colors;
while (*ppColor)
{
        retVal = XAllocNamedColor(dpy,cmapID,*ppColor,&defC,&exactC);
        if (!retVal)
        {
                fprintf(stderr, "Could not allocate a color (\"%s\")\n",*ppColor);
                exit(1);
        }
        ppColor++;
}

wAttr.event_mask = ExposureMask;
wAttr.border_pixel = WHITE;
wAttr.background_pixel = BLACK;
wAttr.colormap = cmapID;
XFlush(dpy);
w = XCreateWindow(dpy,
                  RootWindowOfScreen(ScreenOfDisplay(dpy,0)),
                  0, 0,
                  charWidth * WIDTH, charHeight * HEIGHT,
                  0, DEPTH, CopyFromParent,
                  pVisInfo->visual,
                  CWBackPixel | CWColormap | CWBorderPixel | CWEventMask,
                  &wAttr);
if (!w)
{
fprintf(stderr,"Could not create a window\n");
exit(1);
}

/*
 * Now that the window is created, we need to map it.  Notice
 * that we did not install the colormap that we created.  That
 * is not our job.  That should be left to the window manager
 * to do under whatever policy it chooses.
 */

XMapRaised(dpy,w);
XFlush(dpy);

/*
 * To render, we will need a graphics context of the proper
 * depth.
 */

gc = XCreateGC(dpy, w, 0, NULL);
```

```
        /*
         * We will not set the appropriate values that do not match
         * the defaults.
         */

        XSetFont(dpy, gc, myFontStruct->fid);
        XSetForeground(dpy, gc, WHITE);
        XSetBackground(dpy, gc, BLACK);


        /*
         * Now we'll go into a loop waiting for the next event.  The
         * only event that we've expressed interest in is expose, so
         * when we get one, we'll just refresh the window.
         */

        while(1)
        {
            XNextEvent(dpy, &myEvent);

            /* Put up HEIGHT rows of WIDTH characters on the window */

            for( i = 0; i < HEIGHT; i++ )
            {
                yPos = i * charHeight + myFontStruct->max_bounds.ascent;

                XDrawImageString(dpy, w, gc, 0, yPos, BIG_STRING, WIDTH);
                XFlush(dpy);

            }

        }

}
```

HP OSF/Motif Window Manager Programmatic Interface

H

This chapter discusses the following topics:

- MWM Programmatic Interface Standards.
- Inter-Client Communication Conventions.

H.1 MWM Programmatic Interface Standards

The OSF/Motif Window Manager programmatic interface is based on the **Inter-Client Communications Conventions Manual** (ICCCM ed. December, 1988). The ICCCM establishes the standards for "good citizenship" among clients in a multi-client environment. To avoid costly compatibility problems, you should design and code your client application to operate as a "good citizen."

Since the interaction of your client with MWM occurs primarily as a result of Xlib, Xt Intrinsics, and Xm Widget calls, and some versions of Xlib do not completely support the December 1988 ICCCM, if your client application uses Xlib calls, make sure those calls are supported by the December 1988 ICCCM.

The HP OSF/Motif Window Manager fully supports the December 1988 edition of the ICCCM. Earlier editions of the ICCCM are supported only to the extent that it is necessary to handle clients that use R2 and R3 versions of the X11 Xlib and Xt Intrinsics libraries.

H.2 Inter-Client Communication Conventions

The ICCCM section "Client to Window Manager Communication" specifically discusses how clients communicate with a window manager. Reading the section is recommended. It will give you generally applicable information about how your client application should communicate with a window manager. The remainder of this chapter provides you with additional client information and MWM specific information.

H.2.1 Programming Client Actions

As mentioned above you should design your client application to be a good citizen whether or not a window manager is present to police the environment. The following information will help you program your client application to be a good citizen in a multi-client environment.

Creating a Top-Level Window

The typical way to create a top-level window for your client is as a child of the root window using a call to the Xlib function `XCreateSimpleWindow()`.
However, when you create a window using `XCreateSimpleWindow()`, you must set up your client using properties such as `XStoreName` and calls to the appropriate `XSet*` functions.

The recommended alternative to creating a top-level window with `XCreateSimpleWindow()` is to use the Xt Intrinsics function `XtCreateWindow()`.

At any time, the top-level windows of your client application have one of three states:

Normal	A normal application window is displayed.
Iconic	An icon window is displayed instead of a normal window.
Withdrawn	No normal or iconic window is displayed.

Working with Client Properties

Each top-level window you create for your client should have a list of properties associated with it. These properties are what the window manager inspects to determine how it should manage the client's behavior.

This is especially important in the case where the proper operation of your client application depends on particular property values: Any properties you *don't* specify are specified by the window manager *using whatever values are most convenient.*

Client applications have the following properties:

WM_NAME.

The WM_NAME property contains a string to be displayed in the title area of the client window frame. MWM can dynamically change the window title if your client application changes the value of the string in the WM_NAME property.

If you don't set the WM_NAME property, MWM looks for a title in the `res_name` part of the WM_CLASS property. If MWM finds no title, it uses the string "*****" as the window title.

The window manager assumes that the string passed in the WM_NAME property is compatible with the font used for the window title.

WM_ICON_NAME

The WM_ICON_NAME property contains a string to be displayed in the label part of the icon that is associated with the client window. MWM can dynamically change the icon label if the WM_ICON_NAME property value is changed by the client.

If you don't set the WM_ICON_NAME property, MWM uses the window title as the icon label.

The window manager assumes that the string passed in the WM_ICON_NAME property is compatible with the font used for the icon label.

WM_NORMAL_HINTS

The WM_NORMAL_HINTS property contains a list of fields. MWM tracks changes to the WM_NORMAL_HINTS property. Changes affect *subsequently* created clients. That is, existing clients remain unaffected by changes to WM_NORMAL_HINTS.

The WM_NORMAL_HINTS property contains the following fields:

flags

MWM places windows on the screen using configuration information on size and position (location). The order of precedence MWM uses to look for this information is as follows:

User specified. The client has been supplied configuration information by the user, using USSize and USPosition in the `/Xutil.h` header file.

Interactive placement. Interactive placement is established with the `interactivePlacement` resource (see Chapter 4).

Default configuration.

min_width, min_height

The values set for minimum width and minimum height are used to configure a minimum client size window. If the values set for these fields are are not greater than 0, or not set at all, then a value of 1x1 or larger is used by MWM. The actual minimum size used by MWM is based on the window size that fits in the minimum frame size for the frame type that is being used.

max_width, max_height

The values set for maximum width and maximum height are used only if the `maximumClientSize` resource is not configured. The values set with these fields are used to set a maximum client size window. If max_width and max_height are not configured, then MWM will size the window and its frame to exactly fill the screen. The maximum size of a window can be limited by the `maximumMaximum` resource. (See Chapter 4 for resource descriptions.)

width_inc, height_inc

The values set for width increase and height increase determine the unit of measure used to report window size. When windows are being resized, a feedback window reports the current size in the units specified. If values are not set for these fields, then 1 pixel is used as the sizing increment.

min_aspect.x, min_aspect.y

The values set for minimum aspect.x (width) and minimum aspect.y (length) determine constraints for the minimum ratio of width/length of a window. MWM will apply a minimum aspect ratio sizing constraint when the x and y values are set greater than or equal to zero. The values must also be less than or equal to the max_aspect values.

max_aspect.x, max_aspect.y

The values set for maximum aspect.x (width) and maximum aspect.y (length) determine constraints for the maximum ratio of width/length of a window. MWM will apply a maximum aspect ratio sizing constraint when the x and y values are set greater than or equal to zero. The values must also be greater than or equal to the min_aspect values.

base_width, base_height

The values set for these fields determine the amount of "padding" (margin) between the window and the window frame. The base width value sets the amount of left and right padding. The base height value sets the amount of top and bottom padding. If these fields have a value of less than 0, or if there is no value set, then MWM uses a value of 0.

WM_HINTS

The WM_HINTS property contains a list of fields. Except for changes to the icon_pixmap, MWM tracks changes to the WM_HINTS property only when the client window changes state from the withdrawn state to the normal or iconic state.

The WM_HINTS property contains the following fields:

icon_pixmap	Image for icon window.
icon_window	A working window for the icon window.
icon_x	X coordinate for icon window position.

icon_y	Y coordinate for icon window position.
icon_mask	MWM does not use this.
input	MWM does not use this.
window_group	MWM does not use this.

WM_CLASS

The WM_CLASS property contains two fields. MWM tracks changes to the WM_CLASS property only when the client window changes state from the withdrawn state to the normal or iconic state.

The res_class and res_name values are used by MWM to do client specific configuration of window decorations and icons. If the WM_CLASS property is not set, then no special client customization will be done.

The WM_CLASS property contains the following fields:

res_class	When a client enters MWM's management, the window manager looks at the res_class value to determine the client's class. All resources previously configured for that class will be used for the new client.
res_name	When a client enters MWM's management, the window manager looks at the res_name value to determine the name to use in the client's window title. This field's value is used when the WM_NAME property is not set.

WM_TRANSIENT_FOR

MWM regards a transient window as equivalent to a secondary window. A transient window is always on top (in terms of stacking order) of its primary window. This primary window is identified by the WM_TRANSIENT_FOR property.

The window manager places transient windows on the screen without user interaction. MWM determines window size and placement based on previously specified resource values. The amount of decoration for a transient window is controlled by the `transientWindow` resource. (See Chapter 4)

A transient window is normally associated with a primary window. You can design your client windows such that transient windows are arranged in a tree structure where a transient window has another transient window as its associated "primary" window. However, the root of the tree must be a non-transient window.

WM_PROTOCOLS

The WM_PROTOCOLS property contains a list of atoms (32-bit values that represent unique names). Each atom identifies a protocol in which the client is willing to participate. Atoms can identify standard protocols and private protocols specific to individual window managers. MWM tracks changes to the WM_PROTOCOLS property and supports the following standard protocols:

WM_DELETE_WINDOW

Clients are notified when the MWM `f.kill` function is invoked by the user. MWM does not terminate the client or destroy the window when a WM_DELETE_WINDOW notification is done.

WM_SAVE_YOURSELF

Clients with this atom will be notified when a session manager or a window manager wishes the window's state to be changed. The typical change is when the window is about to be deleted or the session terminated.

quitTimeout.

The `quitTimeout` resource specifies the amount of time (in milliseconds) that MWM will wait for a client to update the WM_COMMAND property after it has sent the WM_SAVE_YOURSELF message. This protocol will only be used for those clients that have a WM_SAVE_YOURSELF atom in the WM_PROTOCOLS client window property. The default time is 1000 (ms).

WM_TAKE_FOCUS

Clients with this atom will be notified when a window manager believes that the client should explicitly set the input focus to one of its windows.

_MOTIF_WM_MESSAGES

Clients with this atom will indicate to the window manager which messages (sent by the window manager when the `f.send_msg` function is invoked) are currently being handled by the client.

WM_COLORMAP_WINDOWS

The WM_COLORMAP_WINDOWS property indicates to MWM which colormaps your client application would like to have installed.

Working with Window Manager Properties

MWM uses properties to supply configuration and state information to clients (usually session managers).

WM_STATE

The WM_STATE property contains the following fields:

state NormalState, IconicState, and WithdrawnState are the values defined for MWM.

icon The icon window value is set to the window ID of the top-level icon window; this window is NOT the icon window supplied by the client. (The icon window, if it is set in WM_HINTS, is a child of the top-level window.)

The information in the WM_STATE property is generally used only by session management clients.

WM_ICON_SIZE

MWM sets the WM_ICON_SIZE property of the root window. WM_ICON_SIZE contains the following fields:

min_width, min_height

Minimum width and minimum height of an icon window are set based on the value of (or default value for) the `iconImageMinimum` resource.

max_width, max_height

Maximum width and maximum height of an icon window are set based on the value of (or default value for) the `iconImage Maximum` resource.

width_inc, height_inc

The increment for changing the width and height of an icon window is set to 1 pixel by MWM.

Changing Window State

Windows are normal (full sized), iconic (small symbol), or withdrawn (not visible). You can control many attributes of normal and icon windows. See Chapter 4 for information on the appearance and behavior of windows in the NormalState. See Chapter 6 for information on the appearance and behavior of windows in the IconicState.

Configuring the Window

Clients can request to be notified, with `ConfigureNotify` events, when windows change size or position. The X,Y coordinates in these events may be relative to either the root window or the frame provided by MWM. Use `XTranslateCoordinates` to determine absolute coordinates.

Changing Window Attributes

If the client requests save-under with the `saveUnder` resource, MWM will set this attribute for the MWM frame instead of the client window.

Controlling Input Focus

Use the `keyboardFocusPolicy` resource to control the input focus. Clients can request to be notified when given the input focus. See "WM_PROTOCOLS."

Windows that supply a WM_PROTOCOLS property containing the WM_TAKE_FOCUS atom will receive a ClientMessage from the window manager.

Establishing Colormaps

If more than one colormap is needed for client subwindows, then set the WM_COLORMAP_WINDOWS property to the list of windows with colormaps.

H.2.2 Client Responses to MWM Actions

MWM redirects the following top-level window requests: `MapWindow`, `ConfigureWindow`, `CirculateWindow`.
MWM may not immediately execute (or execute at all) redirected requests.

Window Size and Position

Clients can request sizes and positions with MWM_HINTS, but MWM may not satisfy these requests.

Window and Icon Mapping

Client windows in the normalized state are mapped. Client windows in the iconified state are not mapped.

Colormap Changes

Clients can request to be notified when their colormap is in use (or no longer in use), by using `ColormapNotify`.

Input Focus

Distribution of input within a client window can be handled using Xt Intrinsics and the Xm Widgets. Clients should generally avoid using XSetInputFocus().

ClientMessage Events

Clients can't prevent being sent `ClientMessage` events, but clients can ignore these if they aren't useful.

H.3 MWM Specific Information

The following information details window manager conventions not covered by the ICCCM, but which are required for supporting HP OSF/Motif behavior.

H.3.1 _MOTIF_WM_HINTS

A client may communicate certain preferences directly to MWM via the **_MOTIF_WM_HINTS** property. The contents of this property is shown in the following table:

Field	Type
flags	CARD32
decorations	CARD32
functions	CARD32
input_mode	CARD32

flags

The flags field indicates which fields in the **_MOTIF_WM_HINTS** property contain data. The following values are supported:

Name	Value	Field
MWM_HINTS_FUNCTIONS	1	MWM functions applicable to client
MWM_HINTS_DECORATIONS	2	Client window frame decorations
MWM_HINTS_INPUT_MODE	4	Client input mode

functions

The functions field indicates which MWM functions should apply to the client window (for example, whether the window should be resized). The information in this field is combined with the value of the `clientFunctions` resource. Function selection using **MWM_HINTS** takes precedence over function selection with the `clientFunctions` resource. Also, decorations that support a particular function (for example, the minimize button) will not be shown if the associated function is not applicable.

Name	Value	Comments
MWM_FUNC_ALL	1	If set, remove functions from full set
MWM_FUNC_RESIZE	2	f.resize
MWM_FUNC_MOVE	4	f.move
MWM_FUNC_MINIMIZE	8	f.minimize
MWM_FUNC_MAXIMIZE	16	f.maximize
MWM_FUNC_CLOSE	32	f.kill

decorations

The decorations field indicates how the client window frame should be decorated (for example, whether the window should have a title bar or window menu button). The information in this field is combined with the value of the `clientDecoration` resource (see Chapter 4,"Using Frameless or Reduced-Element Window Frames"). Decoration selection using **_MOTIF_WM_HINTS** takes precedence over decoration selection with the `clientDecoration` resource.

The following values are supported:

Name	Value	Comments
MWM_DECOR_ALL	1	If set, remove decorations from full set
MWM_DECOR_BORDER	2	Client window border
MWM_DECOR_RESIZEH	4	Resize border handles
MWM_DECOR_TITLE	8	Title bar
MWM_DECOR_SYSTEM	16	Window menu button
MWM_DECOR_MINIMIZE	32	Minimize window button
MWM_DECOR_MAXIMUM	64	Maximize window button

input_mode

The input_mode field indicates the keyboard input focus constraints that are imposed by the client window.

Name	Value	Comments
INPUT_APPIPCATION_MODAL	1	Input does not go to the primary window
INPUT_SYSTEM_MODAL	2	Input goes only to this window

_MOTIF_WM_MENU

The client uses the _MOTIF_WM_MENU property to add menu items to the end of the window menu for the client window. The contents of the property are a list of lines separated by the new line characters `\n`, with the following format:

label [*mnemonic*] [*accelerator*] *function* \n *label* [*mnemonic*] [*accelerator*] *function*

The interpretation of the strings is the same as for menu items (see Chapter 5, "Making New Menus - Menu Items").

_MOTIF_WM_MESSAGES

The client uses the _MOTIF_WM_MESSAGES property to indicate to the window manager which messages (sent by the window manager when the `f.send_msg` function is invoked) are currently being handled by the client. Menu items that have `f.send_msg` specified as the function have grayed-out labels when the associated message is not being handled by the client.

This client property is tracked by the window manager if the _MOTIF_WM_MESSAGES atom is included in the client's WM_PROTOCOLS property. The _MOTIF_WM_MESSAGES property contains a list of integers (in the XChangeProperty: type atom is INTEGER, format is 32). A client places the property on a client window and it is processed by MWM when the client window goes from withdrawn state to normalized or iconified state. Changes to the property are processed while the client window is not in the withdrawn state.

_MOTIF_WM_INFO

The client receives MWM-specific information via the _MOTIF_WM_INFO property. This property is placed by MWM on the root window and is used by clients. The _MOTIF_WM_INFO property is set up as part of MWM initialization. The contents of the _MOTIF_WM_INFO property are shown in the following table.

Field	Type
flags	CARD32
wmWindow	CARD32

flags. The following values can be used alone, or together (using the Boolean "OR").

Name	Value	Field
MWM_INFO_STARTUP_STANDARD	1	Set for startup with standard behavior.
MWM_INFO_STARTUP_CUSTOM	2	Set for startup with customized behavior.

wmWindow. The wmWindow field is always set to the window "ID" of a window that is used by MWM. When MWM is running, the _MOTIF_WM_INFO property is present on the root window and wmWindow is an ID for a window that exists.

H.3.2 Window Management Calls

Clients communicate with the window manager through properties associated with top-level windows, synthetic events (generated using XSendEvent) and standard X events. Programmatically this communication involves Xlib calls (directly or through libraries such as Xt Intrinsics). Clients may programmatically interact with MWM (or any X11 window manager) in one of the following ways:

- **No explicit programmatic access.**

In this case, clients do not set up any window properties or do any call that directly communicates to the window manager. Communication occurs (indirectly) when the state of the client window is changed (that is, the window is mapped, unmapped, configured, has a colormap change, etc.). To work with MWM, clients are not required to do anything more than what is required when a window manager is not being used.

- **High-level programmatic access.**

  Clients can make high-level Xlib call (XSetStandardProperties) or calls to certain libraries built on Xlib (Xt Intrinsics - XtInitialize, XtMainLoop) to establish and maintain standard communications with MWM. Client developers are encouraged to use the X Toolkit for client/window manager communication unless the client has some specialized window management requirements.

- **Low-level programmatic access.**

  Clients with special window management requirements can use low-level Xlib calls (XStoreName, XSetWMHints, etc.) to communicate with the window manager.

Xlib Calls

The calls in the following table are used with MWM:

This Xlib call...	Does this...
XSetStandardProperties()	Sets WM_NAME, WM_ICON_NAME, WM_HINTS, WM_COMMAND, and WM_NORMAL_HINTS. It does not set WM_CLASS (which should be set to allow MWM to be optimally configured for a particular class of client windows).
XStoreName()	Sets the WM_NAME property (used for window titles).
XSetIconName()	Sets the WM_ICON_NAME property (used for the icon label).
XSetCommand()	Sets the WM_COMMAND property.
XSetWMHINTS()	Sets the WM_HINTS property.
XSetNormalHints()	Sets the WM_NORMAL_HINTS property.
XGetIconSizes()	Is used to get a list of MWM-supported icon sizes.
XSetClassHint()	Is used to set the WM_CLASS property.
XSetTransientForHint()	Sets the WM_TRANSIENT_FOR property.
XGetStandardColorMap()	Is used to get standard colormap information.
XSetproperty	

Xt Intrinsics Calls

The calls in the following table are used with MWM:

This Xt Intrinsics call...	Does this...
XtInitialize()	Makes a top-level window and sets up the following properties on that window: WM_NAME, WM_NORMAL_HINTS, WM_HINTS, and WM_CLASS.
XtMainLoop()	Handles the messages described in the ICCCM that deal with window reconfiguration.

Fortran Bindings I

Since X11 is are implemented in the programming language "C", a number of programming techniques have been used that do not have direct analogs in standard Fortran, or even in the HP extensions to Fortran.

For example, standard Fortran passes all parameters by reference. That is, a pointer to the parameter is passed rather than the parameter itself. This is true even for literal constants. Because the state of a window in X is a complicated grouping of dissimilar types, C structures are used to represent them.

As a solution to the problem, ten routines have been developed to create, manage and destroy the data types necessary to call routines in X11. The objects created by these routines can be passed directly to X11.

To allow for maximum flexibility and extensibility, two more routines are provided to add or replace types in the type tables.

All routines not explicitly returning a value are logical functions. A "FALSE" return value implies failure – the failure type is in *xfErrno*. (See the discussion of *xfErrno* in *XfPack*, below).

In order to access Xflib, a program must contain the following statement at the beginning of the file: *include '/usr/include/Xf11/Xfalias.h'*, and the following statement at the beginning of each subprogram wanting to use libXf: *include '/usr/include/Xf11/Xflib.h'*.

I.1 Translating C types to Fortran

The simple types in C have the following correspondence to types in Fortran:

C Types	FORTRAN Types
char	CHARACTER
short	INTEGER*2
int	INTEGER*4
long	INTEGER*4
float	REAL*4
double	REAL*8

In C, variables are declared by specifying the type followed by the variable name. If the variable is to be a pointer, an asterisk is placed between the type name and the variable name. Two asterisks would imply a pointer to a pointer and each succeeding asterisk implies another level of indirection.

Examples:

```
char  fname;
int   *width, *height;
short **data;
```

A structure in C, called "struct", is a grouping of items of dissimilar types. Structs are distinct from arrays in that arrays must contain one or more items of a single type. The typical use of Fortran bindings is to fill in a C structure that will be passed in a call to X11, or to read a C structure returned from a call to X11.

The various items that are contained in a struct are called fields. To access a field of a struct in C, one specifies the struct name, followed by a period, followed by the field name. When using the Fortran bindings, accessing the fields of a struct is done via calls to the `XfInsert()` and `XfExtract()` routines (routines referenced in this section are discussed in detail in the following sections) for assignment to the field and assignment from the field respectively.

Any struct used by X11 may be filled in or read by `XfInsert()` or `XfExtract()`. Whenever the C documentation contains a line like:

this_struct.this_field = this_value;

the Fortran equivalent would be:

XfInsert(XFT_this_struct,XFF_this_field,this_value)

Note that any struct name in the C documentation is preceded by "XFT_" (X/Fortran Type) and any field name is preceded by "XFF_" (X/Fortran Field). The X11 struct and field names are given constant numeric values in the include files "xftypes.h" and "xffields.h" respectively.

Often C structs will contain embedded structures or arrays. Inserting or extracting values from these embedded aggregates is the purpose of the routine `XfAttach()`. By attaching to a field of a structure created by `XfCreate()`, one can insert or extract values from the fields or elements of the embedded aggregate. The common use for this feature is to insert strings into an array of strings or a pointer to an array of strings.

Another use for the `XfAttach()` routine is to allow direct access to pointers. The Fortran bindings will assume that if a field is a pointer, the caller is passing a pointer generated by a previous call to an X11 function. The only exception to this rule is if the pointer being passed is a pointer to a char, i.e., the pointer is a string.

At times one may wish to pass a string generated by an X11 call,or one may wish to generate a pointer to a Fortran variable. This can be done by attaching to the pointer and indexing it in the `XfInsert()` call. When a pointer is indexed by 0, the bindings will assume the caller is speaking of the pointer itself and will pass a pointer value; when a pointer is indexed by one, the bindings will assume the caller wishes the pointer to point to the value being passed. If a pointer is indexed by more than 1, the bindings will assume the caller wishes to point to a list of items and will allocate space for the list and place the value passed at the specified index in the list.

For example:

```
      INTEGER*4 string,ptr

C Place a string in a Fortran bindings variable (XfPack defaults to a
C field of 1)
      string = XfPack(XFT_STRING8,'Some string')
C Get a pointer to the string to pass to a function (by indexing by 0)
      ptr = XfValue(string,0)
```

I.2 Creating an X11 Object

Three routines are available for creating an object to be used by X11: `XfCreate()`, `XfPack()`, and `XfUnpack()`.

I.2.1 XfCreate

The function `XfCreate` (*object_type*) creates an object of the type specified by the parameter *object_type*. *Object_type* is a unique identifying integer assigned to each data type required by X11. These identifying integers are defined in an include file named "xftypes.h" (which is included by Xflib.h) which must be included into any Fortran program using these bindings.

All fields of any objects created via `XfCreate()` will be initialized to zeros. The value returned from the function may be passed to X11 in lieu of a pointer.

Pointers to existing objects are indicated in C by a leading ampersand ("&"). Pointers are declared with a leading asterisk ("*").

For example:
struct sttype st;

```
    .
    .
    .
```

ThisRoutine(&st)
or
struct sttype *stp;

```
    .
    .
    .
```

ThisRoutine(stp)
would both pass a pointer to a structure of type "sttype".

I.2.2 XfPack

The function `XfPack` (*object_type,val1,val2,...,valn*) creates an object in a fashion similar to `XfCreate()`. `XfPack()`, however, will fill in the fields of the created object from the list of values provided. The list of values must be presented in the same order as found in the structure and all values must be supplied.

I.2.3 XfUnpack

The function `XfUnpack` (*object_type,var1,var2,...,varn*) will extract all the variables from the object indicated by *object_type* into the series of variables given. The list of variables must be presented in the same order as found in the structure and all variables must be supplied.

If any one of `XfCreate()`, `XfPack()`, or `XfUnpack` fail, a zero value is returned and an error code is placed in an external variable named *xfErrno*. The error codes are the following:

1. XFE_TOOBIG: too many types have been declared.
2. XFE_NOMEM: out of memory.
3. XFE_BADTYPE: a blatantly illegal type was passed to a routine.

4. XFE_NOTFOUND: either a type (XFT_) or field name (XFF_) was passed to a routine and the type could not be found in the type tables, or the type was found and did not contain the field.

5. XFE_INTERNAL: an internal error was discovered. This usually means that the type tables have been corrupted by a bad call to `XfAddType()` or `XfReplaceType()`.

I.2.4 Examples

```
INTEGER*4 MYSTRUCT
MYSTRUCT = XfCreate(XFT_RECTANGLE)

INTEGER*4 MYSTRUCT
INTEGER*4 x,y,width,height
MYSTRUCT = XfPack(XFT_RECTANGLE,50,50,50,50)
IF (.NOT. XfUnpack(XFT_RECTANGLE,x,y,width,height)) CALL error
```

I.3 Managing Objects

Six routines have been provided to manage the contents of X11 objects. These are `XfInsert()`, `XfExtract()`, `XfValue()`, `XfAttach()`, `XfDetach()`, and `XfSync()`.

I.3.1 XfInsert

After creating an X11 object via either `XfCreate()` or `XfPack()`, values may be placed into fields of the object by the routine `XfInsert` (*Object_ID,Field_ID,value*). *Object_ID* is the return value from a previous call to `XfCreate()` or `XfPack()`, *Field_ID* is a unique identifying integer for a field of the object as defined in the header file "xffields.h" (which is included by Xflib.h) and *value* is the value to be placed in that field.

If a field is described as being a pointer (e.g., "char *"), it may be considered as pointing to an array of items. In the simplest case, the array pointed to has a single element, a pointer. All arrays are indexed starting at one. If a pointer is indexed by zero, the insert and extract functions will assume the user is talking about the pointer itself, rather than the item pointed to. The insert and extract functions will default to an index of zero for all pointers except pointers to characters. Since, in C, pointers to characters are used to pass strings, pointers to characters are assumed to be indexed by one (see the example on pointers in the second section of this appendix). The routine `XfAttach()`, to be described later, allows the user to override these defaults.

Strings and simple types will default to an index of 1. Complex types (e.g., structures) will default to an index of 0. Indexing a simple type by zero will return the X/Fortran version of the variable and is therefore a simple way to generate a pointer to a simple (scalar) type.

I.3.2 XfExtract

`XfExtract` (*Object_ID,Field_ID,value*) is the inverse of `XfInsert()`. `XfExtract()` is used to move a value from a field of an X11 object to a Fortran variable.

I.3.3 XfValue

The function `XfValue` (*objId,fieldId*) extracts a value from the object "objId" in field "fieldId". If the value is a simple (scalar) type (e.g., int or char), enumerated type, or pointer, the value returned will be the actual value extended to be an INTEGER*4. If the value is a complex type (e.g. struct or array), the value returned will be a pointer to the object.

If fieldId is zero, the behavior is similar to the behavior of `XfExtract`.

ObjId must be an object identifier created via `XfCreate()`, `XfPack()` or `XfAttach()`.

I.3.4 XfAttach

`XfAttach` (*Object_ID,Field_ID,Old_Attach_ID*) is a function returning another object identifier. This new identifier is an object whose value is the field specified by *Field_ID*. The object returned is suitable for passing to calls to `XfInsert()`, `XfExtract()` or a subsequent call to `XfAttach()`.
If *Old_Attach_ID* is zero, a new object will be created – if *Old_Attach_ID* is non-zero, and is an object identifier created via a previous call to `XfAttach()`, it will be re-used. It is an error to provide an *Old_Attach_ID* that is non-zero but was not created by a call to `XfAttach()`. `XfAttach()` is particularly useful for filling in structures with embedded structures or arrays. By attaching to the inner structure, one can avoid the creation of an intermediate structure for filling in the values.

Another use of `XfAttach()` is to allow indexing of pointers. By attaching to the pointer, the user can specify the index when inserting or extracting. This allows the user to insert a character pointer returned from an X11 call directly into a structure (by specifying an index of zero), or a pointer to an item to be generated (by specifying an index of one.)

I.3.5 XfDetach

`XfDetach` (*Object_ID*) releases the temporary object identifier created by a previous call to `XfAttach()`. It is an error if *Object_ID* was not created by a call to `XfAttach()`.

I.3.6 XfSync

`XfSync()` guarantees that the X/Fortran version of certain global X11 variables are up to date. It should be used before accessing the following variables after X calls:

Variable name	Type
_xfCurrentDisplay	XFT_Display
xfZeroPt	XFT_POINT
xfZeroRect	XFT_RECTANGLE
xfBaseFontInfo	XFT_XFontStruct
xfCursorImage	XFT_INT16Pointer
xfCursorMask	XFT_INT16Pointer
xf_bitmaps	XFT_INT16x16Pointer (pointer to array of 16 16 bit integers)
xf_PolyList	XFT_XPointPointer

I.4 Releasing an Object

To avoid consuming memory without bound, a routine has been provided to release the memory claimed by a call to `XfCreate()` or `XfPack()`. This routine, `XfDestroy` (*Object_ID*), returns any memory used to hold the values of the object referred to by *Object_ID* to the available memory pool. It is an error if *Object_ID* was not created by a previous call to `XfCreate()` or `XfPack()`.

I.4.1 Example

```
INTEGER*4 NEWSIZE

NEWSIZE = XfCreate(XFT_RECTANGLE)
.
.
.
CALL XfDestroy(NEWSIZE)
```

I.5 Extending the Fortran Bindings

In some instances a programmer will need to extend the bindings to describe a type that may only occasionally be used. Two functions, `XfAddType` (*Type_ID,Descriptor*) and `XfReplaceType` (*Type_ID,Descriptor*) allow new types to be added to the Fortran binding software. *Type_ID* is a unique identifying integer by which the type will be known (or zero to allow the bindings to create an appropriate identifier), and is the value that would be passed to `XfCreate()` or `XfPack()`. The *Descriptor* is the means by which the size and contents of the type are specified. The return value of the call is the newly created type or zero if the call fails.

The fields are passed in as a two dimensional array of integers in Fortran and can be thought of as an array of pairs. The first pair of each descriptor must contain one of the following values:

(XFT_pointer,0)
(XFT_enum,0)
(XFT_array,0)
(XFT_union,0)
(XFT_struct,0)

For pointers, the pairs describe the type pointed to. For example, a pointer to an integer would be described by the pairs:

```
((XFT_pointer,0),(XFT_int,0)).
```

For pointers, the values supplied to `XfPack()` must be variables, not constants—except that you *can* use string constants.

To create and use the above described pointer to an integer, the following descriptor would be passed:

```
    DATA ((integerPointerFields(j,k),j=1,2),k=1,2)
   C/XFT_pointer,0,XFT_int,0/
```

This is illustrated in the following example:

```
    INTEGER*4 newType,newValue,integerPointerFields(2,2)
    INTEGER j,k,l
    DATA ((integerPointerFields(j,k),j=1,2),k=1,2)
   C/XFT_pointer,0,XFT_int,0/
    .
    .
    .
    newType = XfAddType(0,integerPointerFields)
    .
    .
    .
    l = 10
    newValue = XfPack(newType,l)
    .
    .
    .
C specifying a field of 1 to XfValue retrieves the value pointed to
                IF (XfValue(newValue,1) .EQ. 10) CALL ...
```

The first pair of an enumerated type descriptor consists of the values (XFT_enum,0). In succeeding pairs, the first element of each holds the external value of the field. The second element holds the symbolic identifier by which the value will be known. The end of the list of enumerated types is indicated by a field identifier of zero. An enumerated type consisting of the possible values: Name1, Name2 and Name3 would be described by the pairs: ((XFT_enum,0),(0,Name1),(1,Name2),(2,Name3),(0,0)).

An example of the creation and use of such an enumerated type is:

```
    INTEGER*4 newType,newValue,myEnumeratedFields(2,5)
    INTEGER j,k
    DATA ((myEnumeratedFields(j,k),j=1,2),k=1,5)
   C/XFT_enum,0,0,Name1,1,Name2,2,Name3,0,0/
    .
    .
    .
    newType = XfAddType(0,myEnumeratedFields)
    .
    .
    .
    newValue = XfPack(newType,Name2)
    .
    .
    .
   C specifying a field value of 1 retrieves the symbolic value
                IF (XfValue(newValue,1) .EQ. Name2) CALL ...
```

Arrays are described starting with a pair consisting of (XFT_array,0). The following pairs first describe the base type of the array followed by a pair consisting of the number of elements in the array and a zero. To create and use a type describing a 2 element array of items of type integer, one would enter:

```
    INTEGER*4 newType,newValue,myArray2IntegerFields(2,3)
    INTEGER j,k
    DATA ((myArray2IntegerFields(j,k),j=1,2),k=1,3)
   C/XFT_array,0,XFT_int,0,2,0/
    .
    .
    .
    newType = XfAddType(0,myArray2IntegerFields)
    .
    .
    .
    newValue = XfPack(newType,3,4)
    .
    .
    .
C the field value is the index into the array
                  IF (XfValue(newType,2) .EQ. 4) CALL ...
```

Unions allow a variable to be accessed as one of several types. A union descriptor begins with the pair (XFT_union,0) followed by pairs consisting of previously defined types and a field identifier which must be non-zero and unique within the union. To create and use a union of two types, character or integer, one would need the pairs ((XFT_union,0), (XFT_char,n1),XFT_integer,n2),(0,0)), where n1 and n2 are distinct and non zero.

For example, in Fortran:

```
                  PARAMETER (C = 1, I = 2)
                  INTEGER C,I
                  INTEGER*4 newType,newValue,myUnionFields(2,4)
                  INTEGER j,k
                  DATA ((myUnionFields(j,k),j=1,2),k=1,4)


      C/XFT_union,0,XFT_char,C,XFT_int,I,0,0/
                  .
                  .
                  .
                  newType = XfAddType(0,myUnionFields)
                  .
                  .
                  .
                  newValue = XfCreate(newType)
C insert a character 'x' into the union
                  IF (.NOT. XfInsert(newValue,C,'x')) CALL error
```

Finally, structures begin with the pair (XFT_struct,0) followed by a list of fields terminated with a pair having a first element of zero. The first pair of the descriptor of a field type value will have the symbolic name by which the field will be known as its second element. To create a structure consisting of an integer and an array of two characters one would need the pairs: ((XFT_struct,0),(XFT_int,n1),(XFT_array,n2),(XFT_char,0),(2,0)) where n1 and n2 are distinct and non-zero.

Here is an example of declaration and use of such a structure:

```
                  PARAMETER (I = 1, CA = 2)
                  INTEGER C,I
                  INTEGER*4 newType,newValue,attach,myStructFields(2,5)
                  INTEGER j,k
                  DATA ((myStructFields(j,k),j=1,2),k=1,5)
     C/XFT_struct,0,XFT_int,I,XFT_array,CA,XFT_char,0,2,0/
                  .
                  .
                  .
                  newType = XfAddType(0,myStructFields)
                  .
                  .
                  .
                  newValue = XfCreate(newType)
C attach the the array of two characters
                  attach = XfAttach(newType,CA,0)
C insert an 'x' in the second element of the array
                  IF (.NOT. XfInsert(attach,2,'x')) CALL error
```

I.6 FORTRAN/X Program Examples

Following is a program rewritten in FORTRAN.

```
C     Translation of Sample Program 1 taken from chapter 1 of
C     "Programming with the Xrlib User Interface Toolbox"
C
      INCLUDE '/usr/include/Xf11/xfalias.h'
C
C
      PROGRAM sample1
      INCLUDE '/usr/include/Xf11/Xflib.h'
C
      INTEGER*4 display,screen,gc
      INTEGER*4 border,background
      INTEGER*4 windowId
      INTEGER*4 wAttribs
      INTEGER*4 i,j
C
C     Open the display
C

      display=XOpenDisplay(0)
      if (display .ne. 0) goto 10
      print *,'cannot create a window'
      goto 9999

10    screen=DefaultScreen(display)
      border=BlackPixel(display,screen)
      background=WhitePixel(display,screen)

C
C     Create a window and put it on the display
C

      windowId = XCreateSimpleWindow(display,
     C  RootWindow(display,screen),
     C  50,50,400,200,3,border,background)


      wAttribs=XfCreate(XFT_XSetWindowAttributes)
      if (XfInsert(wAttribs,XFF_backing_store,XFD_WhenMapped)) goto 20
      print *,'XfInsert (#1) error ->',xfErrno
      goto 9999

20    call XChangeWindowAttributes(display,windowId,
     C  XFD_CWBackingStore,wAttribs)

      call XMapWindow(display,windowId)
```

```
      gc=XCreateGC(display,windowId,0,0)
C
C     Send "Hello world" to the window
C
      i=XfPack(XFT_STRING8,'Hello World')
      if (i .ne. 0) goto 40
      print *,'XfPack #2) error ->',xfErrno
      goto 9999
40    if (XfExtract(i,0,j)) goto 50
      print *,'XfExtract (#3) error ->',xfErrno
      goto 9999

50    call XDrawString(display,windowId,gc,100,80,j,11)
      call XFlush(display)
      call sleep(5)
      call XCloseDisplay(display)

9999  END
```

NAME

Intro - Introduction to the reference section of the *Programming With Xlib* manual.

DESCRIPTION

This section contains reference information about the C Language functions and macros contained in the Xlib and XHP libraries. Functions are listed in related groups on each manual page.

To locate a particular function use the index that follows. Each routine is listed in alphabetical order followed by the name of the manual page where it is documented.

Function	Location
AllPlanes()	AllPlanes(3X)
BlackPixelofScreen()	BlackPixelofScreen(3X)
ImageByteOrder()	ImageByteOrder(3X)
IsCursorKey()	IsCursorKey(3X)
XActivateScreenSaver()	XSetScreenSaver(3X)
XAddHost()	XAddHost(3X)
XAddHosts()	XAddHost(3X)
XAddPixel()	XCreateImage(3X)
XAddToSaveSet()	XChangeSaveSet(3X)
XAllocColor()	XAllocColor(3X)
XAllocColorCells()	XAllocColor(3X)
XAllocColorPlanes()	XAllocColor(3X)
XAllocNamedColor()	XAllocColor(3X)
XAllowEvents()	XAllowEvents(3X)
XAutoRepeatOff()	XChangeKeyboardControl(3X)
XAutoRepeatOn()	XChangeKeyboardControl(3X)
XBell()	XChangeKeyboardControl(3X)
XChangeActivePointerGrab()	XGrabPointer(3X)
XChangeGC()	XCreateGC(3X)
XChangeKeyboardControl()	XChangeKeyboardControl(3X)
XChangeKeyboardMapping()	XChangeKeyboardMapping(3X)
XChangePointerControl()	XChangePointerControl(3X)
XChangeProperty()	XGetWindowProperty(3X)
XChangeSaveSet()	XChangeSaveSet(3X)
XChangeWindowAttributes()	XChangeWindowAttributes(3X)
XCheckIfEvent()	XIfEvent(3X)
XCirculateSubwindows()	XRaiseWindow(3X)
XCirculateSubwindowsDown()	XRaiseWindow(3X)
XCirculateSubwindowsUp()	XRaiseWindow(3X)
XClearArea()	XClearArea(3X)
XClearWindow()	XClearArea(3X)
XClipBox()	XPolygonRegion(3X)
XCloseDisplay()	XOpenDisplay(3X)
XConfigureWindow()	XConfigureWindow(3X)
XConvertSelection()	XSetSelectionOwner(3X)
XCopyArea()	XCopyArea(3X)
XCopyColormapAndFree()	XCreateColormap(3X)
XCopyGC()	XCreateGC(3X)
XCopyPlane()	XCopyArea(3X)
XCreateBitmapFromData()	XReadBitmapFile(3X)
XCreateColormap()	XCreateColormap(3X)

Function	Location
XCreateFontCursor()	XCreateFontCursor(3X)
XCreateGC()	XCreateGC(3X)
XCreateGlyphCursor()	XCreateFontCursor(3X)
XCreateImage()	XCreateImage(3X)
XCreatePixmap()	XCreatePixmap(3X)
XCreatePixmapCursor()	XCreateFontCursor(3X)
XCreatePixmapFromBitmapData()	XReadBitmapFile(3X)
XCreateRegion()	XCreateRegion(3X)
XCreateSimpleWindow()	XCreateWindow(3X)
XCreateWindow()	XCreateWindow(3X)
XDefineCursor()	XDefineCursor(3X)
XDeleteContext()	XSaveContext(3X)
XDeleteModifiermapEntry()	XChangeKeyboardMapping(3X)
XDeleteProperty()	XGetWindowProperty(3X)
XDestroyImage()	XCreateImage(3X)
XDestroyRegion()	XCreateRegion(3X)
XDestroySubwindows()	XDestroyWindow(3X)
XDestroyWindow()	XDestroyWindow(3X)
XDisableAccessControl()	XAddHost(3X)
XDisplayName()	XSetErrorHandler(3X)
XDrawArc()	XDrawArc(3X)
XDrawArcs()	XDrawArc(3X)
XDrawImageString()	XDrawImageString(3X)
XDrawImageString16()	XDrawImageString(3X)
XDrawLine()	XDrawLine(3X)
XDrawLines()	XDrawLine(3X)
XDrawPoint()	XDrawPoint(3X)
XDrawPoints()	XDrawPoint(3X)
XDrawRectangle()	XDrawRectangle(3X)
XDrawRectangles()	XDrawRectangle(3X)
XDrawSegments()	XDrawLine(3X)
XDrawString()	XDrawString(3X)
XDrawString16()	XDrawString(3X)
XDrawText()	XDrawText(3X)
XDrawText16()	XDrawText(3X)
XEmptyRegion()	XEmptyRegion(3X)
XEnableAccessControl()	XAddHost(3X)
XEqualRegion()	XEmptyRegion(3X)
XEventsQueued()	XFlush(3X)
XFetchBuffer()	XStoreBytes(3X)
XFetchBytes()	XStoreBytes(3X)
XFetchName()	XStoreName(3X)
XFillArc()	XFillRectangle(3X)
XFillArcs()	XFillRectangle(3X)
XFillPolygon()	XFillRectangle(3X)
XFillRectangle()	XFillRectangle(3X)
XFillRectangles()	XFillRectangle(3X)
XFindContext()	XSaveContext(3X)
XFlush()	XFlush(3X)
XForceScreenSaver()	XSetScreenSaver(3X)

Function	Location
XFree()	XFree(3X)
XFreeColormap()	XCreateColormap(3X)
XFreeColors()	XAllocColor(3X)
XFreeCursor()	XRecolorCursor(3X)
XFreeFont()	XLoadFont(3X)
XFreeFontInfo()	XLoadFont(3X)
XFreeFontNames()	XListFonts(3X)
XFreeFontPath()	XSetFontPath(3X)
XFreeGC()	XCreateGC(3X)
XFreeModifierMap()	XChangeKeyboardMapping(3X)
XFreePixmap()	XCreatePixmap(3X)
XGContextFromGC()	XLoadFont(3X)
XGeometry()	XParseGeometry(3X)
XGetAtomName()	XInternAtom(3X)
XGetClassHint()	XSetClassHint(3X)
XGetDefault()	XGetDefault(3X)
XGetErrorDatabaseText()	XSetErrorHandler(3X)
XGetErrorText()	XSetErrorHandler(3X)
XGetFontPath()	XSetFontPath(3X)
XGetFontProperty()	XLoadFont(3X)
XGetGeometry()	XGetWindowAttributes(3X)
XGetIconName()	XSetIconName(3X)
XGetIconSizes()	XSetIconSizeHints(3X)
XGetImage()	XPutImage(3X)
XGetInputFocus()	XSetInputFocus(3X)
XGetKeyboardControl()	XChangeKeyboardControl(3X)
XGetKeyboardMapping()	XChangeKeyboardMapping(3X)
XGetModifierMapping()	XChangeKeyboardMapping(3X)
XGetNormalHints()	XSetNormalHints(3X)
XGetPixel()	XCreateImage(3X)
XGetPointerControl()	XChangePointerControl(3X)
XGetPointerMapping()	XSetPointerMapping(3X)
XGetResource()	XGetResource(3x)
XGetScreenSaver()	XSetScreenSaver(3X)
XGetSelectionOwner()	XSetSelectionOwner(3X)
XGetSizeHints()	XSetSizeHints(3X)
XGetStandardColormap()	XSetStandardColormap(3X)
XGetSubImage()	XPutImage(3X)
XGetTransientForHint()	XSetTransientForHint(3X)
XGetVisualInfo()	XGetVisualInfo(3X)
XGetWindowAttributes()	XGetWindowAttributes(3X)
XGetWindowProperty()	XGetWindowProperty(3X)
XGetWMHints()	XSetWMHints(3X)
XGetZoomHints()	XSetZoomHints(3X)
XGrabButton()	XGrabButton(3X)
XGrabKey()	XGrabKey(3X)
XGrabKeyboard()	XGrabKeyboard(3X)
XGrabPointer()	XGrabPointer(3X)
XGrabServer()	XGrabServer(3X)

Function	Location
XHPAcknowledge()	XHPAcknowledge(3X)
XHPChangeDeviceControl()	XHPChangeDeviceControl(3X)
XHPChangeDeviceKeyMapping()	XHPChangeDeviceControl(3X)
XHPConvertLookup()	XHPConvertLookup(3X)
XHPDeviceAutoRepeatOn()	XHPDeviceAutoRepeatOn(3X)
XHPDeviceAutoRepeatOff()	XHPDeviceAutoRepeatOn(3X)
XHPDisableReset()	XHPDisableReset(3X)
XHPEnableReset()	XHPEnableReset(3X)
XHPFileToPixmap()	XHPFileToPixmap(3X)
XHPFileToWindow()	XHPFileToWindow(3X)
XHPFreeDeviceList()	XHPFreeDeviceList(3X)
XHPGetCurrentDeviceMask()	XHPGetCurrentDeviceMask(3X)
XHPGetDeviceFocus()	XHPGetDeviceFocus(3X)
XHPGetDeviceMotionEvents()	XHPGetDeviceFocus(3X)
XHPGetDeviceControl()	XHPGetDeviceFocus(3X)
XHPGetDeviceKeyMapping()	XHPGetDeviceFocus(3X)
XHPGetDeviceModifierMapping()	XHPGetDeviceFocus(3X)
XHPGetEurasCvt()	XHPGetEurasCvt(3X)
XHPGetExtEventMask()	XHPGetExtEventMask(3X)
XHPGetServerMode()	XHPGetServerMode(3X)
XHPGrabDevice()	XHPGrabDevice(3X)
XHPGrabDeviceButton()	XHPGrabDevice(3X)
XHPGrabDeviceKey()	XHPGrabDevice(3X)
XHPInputChinese_s()	XHPInputChinese_s(3X)
XHPInputChinese_t()	XHPInputChinese_t(3X)
XHPInputISO7sub()	XHPInputISO7sub(3X)
XHPInputJapanese()	XHPInputJapanese(3X)
XHPInputKorean()	XHPInputKorean(3X)
XHPInputRoman8()	XHPInputRoman8(3X)
XHPKeysymToRoman8()	XHPKeysymToRoman8(3X)
XHPListInputDevices()	XHPListInputDevices(3X)
XHPNlioctl()	XHPNlioctl()
XHPPixmapToFile()	XHPPixmapToFile(3X)
XHPPrompt()	XHPPrompt(3X)
XHPQueryImageFile()	XHPQueryImageFile(3X)
XHPSelectExtensionEvent()	XHPSelectExtensionEvent(3X)
XHPSetDeviceFocus()	XHPSetDeviceFocus(3X)
XHPSetDeviceModifierMapping()	XHPSetDeviceFocus(3X)
XHPSetErrorHandler()	XHPSetErrorHandler(3X)
XHPSetInputDevice()	XHPSetInputDevice(3X)
XHPRefreshKeyboardMapping()	XHPSetKeyboardMapping(3X)
XHPSetKeyboardMapping()	XHPSetKeyboardMapping(3X)
XHPUngrabDevice()	XHPUngrabDevice(3X)
XHPUngrabDeviceButton()	XHPUngrabDevice(3X)
XHPUngrabDeviceKey()	XHPUngrabDevice(3X)
XHPWindowToFile()	XHPWindowToFile(3X)
XIfEvent()	XIfEvent(3X)
XInitialize()	XInitialize(3X)
XInsertModifiermapEntry()	XChangeKeyboardMapping(3X)
XInstallColormap()	XInstallColormap(3X)

Function	Location
XInternAtom()	XInternAtom(3X)
XIntersectRegion()	XIntersectRegion(3X)
XKeycodeToKeysym()	XStringToKeysym(3X)
XKeysymToKeycode()	XStringToKeysym(3X)
XKeysymToString()	XStringToKeysym(3X)
XKillClient()	XSetCloseDownMode(3X)
XListFonts()	XListFonts(3X)
XListFontsWithInfo()	XLoadFont(3X)
XListHosts()	XAddHost(3X)
XListInstalledColormaps()	XInstallColormap(3X)
XListProperties()	XGetWindowProperty(3X)
XLoadFont()	XLoadFont(3X)
XLoadQueryFont()	XLoadFont(3X)
XLookupColor()	XQueryColor(3X)
XLookupKeysym()	XLookupKeysym(3X)
XLookupString()	XLookupKeysym(3X)
XLowerWindow()	XRaiseWindow(3X)
XMapRaised()	XMapWindow(3X)
XMapSubwindows()	XMapWindow(3X)
XMapWindow()	XMapWindow(3X)
XMatchVisualInfo()	XGetVisualInfo(3X)
XMergeDataBases()	XMergeDataBases(3X)
XMoveResizeWindow()	XConfigureWindow(3X)
XMoveWindow()	XConfigureWindow(3X)
XNewModifierMap()	XChangeKeyboardMapping(3X)
XNextEvent()	XFlush(3X)
XNoOp()	XFree(3X)
XNoOp()	XOpenDisplay(3X)
XOffsetRegion()	XIntersectRegion(3X)
XOpenDisplay()	XOpenDisplay(3X)
XParseColor()	XParseGeometry(3X)
XParseGeometry()	XParseGeometry(3X)
XPeekEvent()	XFlush(3X)
XPeekIfEvent()	XIfEvent(3X)
XPending()	XFlush(3X)
XPointInRegion()	XIntersectRegion(3X)
XPolygonRegion()	XPolygonRegion(3X)
XPutBackEvent()	XPutBackEvent(3X)
XPutImage()	XPutImage(3X)
XPutPixel()	XCreateImage(3X)
XQueryBestCursor()	XRecolorCursor(3X)
XQueryBestSize()	XQueryBestSize(3X)
XQueryBestStipple()	XQueryBestSize(3X)
XQueryBestTile()	XQueryBestSize(3X)
XQueryColor()	XQueryColor(3X)
XQueryColors()	XQueryColor(3X)
XQueryFont()	XLoadFont(3X)
XQueryKeymap()	XChangeKeyboardControl(3X)
XQueryPointer()	XQueryPointer(3X)
XQueryTextExtents()	XTextExtents(3X)
XQueryTextExtents16()	XTextExtents(3X)

Function	Location
XQueryTree()	XQueryTree(3X)
XRaiseWindow()	XRaiseWindow(3X)
XReadBitmapFile()	XReadBitmapFile(3X)
XRebindKeySym()	XLookupKeysym(3X)
XRecolorCursor()	XRecolorCursor(3X)
XRectInRegion()	XIntersectRegion(3X)
XRefreshKeyboardMapping()	XLookupKeysym(3X)
XRemoveFromSaveSet()	XChangeSaveSet(3X)
XRemoveHost()	XAddHost(3X)
XRemoveHosts()	XAddHost(3X)
XReparentWindow()	XReparentWindow(3X)
XResetScreenSaver()	XSetScreenSaver(3X)
XResizeWindow()	XConfigureWindow(3X)
XRestackWindows()	XRaiseWindow(3X)
XrmPutResource()	XrmPutResource(3X)
XrmUniqueQuark()	XrmUniqueQuark(3X)
XRotateBuffers()	XStoreBytes(3X)
XRotateWindowProperties()	XGetWindowProperty(3X)
XSaveContext()	XSaveContext(3X)
XSelectInput()	XSelectInput(3X)
XSetAccessControl()	XAddHost(3X)
XSetAfterFunction()	XSynchronize(3X)
XSetArcMode()	XSetArcMode(3X)
XSetBackground()	XSetState(3X)
XSetClassHint()	XSetClassHint(3X)
XSetClipMask()	XSetClipOrigin(3X)
XSetClipOrigin()	XSetClipOrigin(3X)
XSetClipRectangles()	XSetClipOrigin(3X)
XSetCloseDownMode()	XSetCloseDownMode(3X)
XSetCommand()	XSetCommand(3X)
XSetDashes()	XSetLineAttribute(3X)
XSetErrorHandler()	XSetErrorHandler(3X)
XSetFillRule()	XSetFillStyle(3X)
XSetFillStyle()	XSetFillStyle(3X)
XSetFont()	XSetFont(3X)
XSetFontPath()	XSetFontPath(3X)
XSetForeground()	XSetState(3X)
XSetFunction()	XSetState(3X)
XSetGraphicsExposure()	XSetArcMode(3X)
XSetIconName()	XSetIconName(3X)
XSetIconSizes()	XSetIconSizeHints(3X)
XSetIconSizeHints()	XSetIconSizeHints(3X)
XSetInputFocus()	XSetInputFocus(3X)
XSetIOErrorHandler()	XSetErrorHandler(3X)
XSetLineAttribute()	XSetLineAttribute(3X)
XSetModifierMapping()	XChangeKeyboardMapping(3X)
XSetNormalHints()	XSetNormalHints(3X)
XSetPlanemask()	XSetState(3X)
XSetPointerMapping()	XSetPointerMapping(3X)
XSetRegion()	XCreateRegion(3X)
XSetScreenSaver()	XSetScreenSaver(3X)

Function	Location
XSetSelectionOwner()	XSetSelectionOwner(3X)
XSetSizeHints()	XSetSizeHints(3X)
XSetStandardColormap()	XSetStandardColormap(3X)
XSetStandardProperties()	XSetStandardProperties(3X)
XSetState()	XSetState(3X)
XSetStipple()	XSetTile(3X)
XSetSubwindowMode()	XSetArcMode(3X)
XSetTile()	XSetTile(3X)
XSetTransientForHint()	XSetTransientForHint(3X)
XSetTSOrigin()	XSetTile(3X)
XSetWindowBackground()	XChangeWindowAttributes(3X)
XSetWindowBackgroundPixmap()	XChangeWindowAttributes(3X)
XSetWindowBorder()	XChangeWindowAttributes(3X)
XSetWindowBorderPixmap()	XChangeWindowAttributes(3X)
XSetWindowBorderWidth()	XConfigureWindow(3X)
XSetWindowColormap()	XCreateColormap(3X)
XSetWMHints()	XSetWMHints(3X)
XSetZoomHints()	XSetZoomHints(3X)
XShrinkRegion()	XIntersectRegion(3X)
XStoreBuffer()	XStoreBytes(3X)
XStoreBytes()	XStoreBytes(3X)
XStoreColor()	XStoreColors(3X)
XStoreColors()	XStoreColors(3X)
XStoreName()	XStoreName(3X)
XStoreNamedColor()	XStoreColors(3X)
XStringToKeysym()	XStringToKeysym(3X)
XSubImage()	XCreateImage(3X)
XSubtractRegion()	XIntersectRegion(3X)
XSync()	XFlush(3X)
XSynchronize()	XSynchronize(3X)
XTextExtents()	XTextExtents(3X)
XTextExtents16()	XTextExtents(3X)
XTextWidth()	XTextWidth(3X)
XTextWidth16()	XTextWidth(3X)
XTranslateCoordinates()	XTranslateCoordinates(3X)
XUndefineCursor()	XDefineCursor(3X)
XUngrabButton()	XGrabButton(3X)
XUngrabKey()	XGrabKey(3X)
XUngrabKeyboard()	XGrabKeyboard(3X)
XUngrabPointer()	XGrabPointer(3X)
XUngrabServer()	XGrabServer(3X)
XUninstallColormap()	XInstallColormap(3X)
XUnionRectWithRegion()	XIntersectRegion(3X)
XUnionRegion()	XIntersectRegion(3X)
XUniqueContext()	XSaveContext(3X)
XUnloadFont()	XLoadFont(3X)
XUnmapSubwindows()	XUnmapWindow(3X)
XUnmapWindow()	XUnmapWindow(3X)
XWarpPointer()	XWarpPointer(3X)
XWriteBitmapFile()	XReadBitmapFile(3X)
XXorRegion()	XIntersectRegion(3X)

NAME

AllPlanes, BlackPixel, WhitePixel, ConnectionNumber, DefaultColormap, DefaultDepth, DefaultGC, DefaultRootWindow, DefaultScreenOfDisplay, DefaultScreen, DefaultVisual, DisplayCells, DisplayPlanes, DisplayString, LastKnownRequestProcessed, NextRequest, ProtocolVersion, ProtocolRevision, QLength, RootWindow, ScreenCount, ScreenOfDisplay, ServerVendor, VendorRelease - Display macros

SYNOPSIS

AllPlanes()

BlackPixel(display, screen_number)

WhitePixel(display, screen_number)

ConnectionNumber(display)

DefaultColormap(display, screen_number)

DefaultDepth(display, screen_number)

DefaultGC(display, screen_number)

DefaultRootWindow(display)

DefaultScreenOfDisplay(display)

DefaultScreen(display)

DefaultVisual(display, screen_number)

DisplayCells(display, screen_number)

DisplayPlanes(display, screen_number)

DisplayString(display)

LastKnownRequestProcessed(display)

NextRequest(display)

ProtocolVersion(display)

ProtocolRevision(display)

QLength(display)

RootWindow(display, screen_number)

ScreenCount(display)

ScreenOfDisplay(display, screen_number)

ServerVendor(display)

VendorRelease(display)

ARGUMENTS

display Specifies the connection to the X server.

screen_number Specifies the appropriate screen number on the host server.

DESCRIPTION

The *AllPlanes* macro returns a value with all bits set to 1 suitable for use in a plane argument to a procedure.

The *BlackPixel* macro returns the black pixel value for the specified screen.

The *WhitePixel* macro returns the white pixel value for the specified screen.

The *ConnectionNumber* macro returns a connection number for the specified display.

The *DefaultColormap* macro returns the default colormap ID for allocation on the specified screen.

The *DefaultDepth* macro returns the depth (number of planes) of the default root window for the specified screen.

The *DefaultGC* macro returns the default GC for the root window of the specified screen.

The *DefaultRootWindow* macro returns the root window for the default screen.

The *DefaultScreenOfDisplay* macro returns the default screen of the specified display.

The *DefaultScreen* macro returns the default screen number referenced in the *XOpenDisplay* routine.

The *DefaultVisual* macro returns the default visual type for the specified screen.

The *DisplayCells* macro returns the number of entries in the default colormap.

The *DisplayPlanes* macro returns the depth of the root window of the specified screen.

The *DisplayString* macro returns the string that was passed to *XOpenDisplay* when the current display was opened.

The *LastKnownRequestProcessed* macro extracts the full serial number of the last request known by Xlib to have been processed by the X server.

The *NextRequest* macro extracts the full serial number that is to be used for the next request.

The *ProtocolVersion* macro returns the major version number (11) of the X protocol associated with the connected display.

The *ProtocolRevision* macro returns the minor protocol revision number of the X server.

The *QLength* macro returns the length of the event queue for the connected display.

The *RootWindow* macro returns the root window.

The *ScreenCount* macro returns the number of available screens.

The *ScreenOfDisplay* macro returns a pointer to the screen of the specified display.

The *ServerVendor* macro returns a pointer to a null-terminated string that provides some identification of the owner of the X server implementation.

The *VendorRelease* macro returns a number related to a vendor's release of the X server.

SEE ALSO

BlackPixelOfScreen(3X11), ImageByteOrder(3X11), IsCursorKey(3X11)

NAME

BlackPixelOfScreen, WhitePixelOfScreen, CellsOfScreen, DefaultColormapOfScreen, DefaultDepthOfScreen, DefaultGCOfScreen, DefaultVisualOfScreen, DoesBackingStore, DoesSaveUnders, DisplayOfScreen, EventMaskOfScreen, HeightOfScreen, HeightMMOfScreen, MaxCmapsOfScreen, MinCmapsOfScreen, PlanesOfScreen, RootWindowOfScreen, WidthOfScreen, WidthMMOfScreen - screen information macros

SYNOPSIS

BlackPixelOfScreen(screen)

WhitePixelOfScreen(screen)

CellsOfScreen(screen)

DefaultColormapOfScreen(screen)

DefaultDepthOfScreen(screen)

DefaultGCOfScreen(screen)

DefaultVisualOfScreen(screen)

DoesBackingStore(screen)

DoesSaveUnders(screen)

DisplayOfScreen(screen)

EventMaskOfScreen(screen)

HeightOfScreen(screen)

HeightMMOfScreen(screen)

MaxCmapsOfScreen(screen)

MinCmapsOfScreen(screen)

PlanesOfScreen(screen)

RootWindowOfScreen(screen)

WidthOfScreen(screen)

WidthMMOfScreen(screen)

ARGUMENTS

screen Specifies a pointer to the appropriate *Screen* structure.

DESCRIPTION

The *BlackPixelOfScreen* macro returns the black pixel value of the specified screen.

The *WhitePixelOfScreen* macro returns the white pixel value of the specified screen.

The *CellsOfScreen* macro returns the number of colormap cells in the default colormap of the specified screen.

The *DefaultColormapOfScreen* macro returns the default colormap of the specified screen.

The *DefaultDepthOfScreen* macro returns the default depth of the root window of the specified screen.

The *DefaultGCOfScreen* macro returns the default GC of the specified screen, which has the same depth as the root window of the screen.

The *DefaultVisualOfScreen* macro returns the default visual of the specified screen.

The *DoesBackingStore* macro returns *WhenMapped, NotUseful,* or *Always,* which indicate whether the screen supports backing stores.

The *DoesSaveUnders* macro returns a Boolean value indicating whether the screen supports save unders.

The *DisplayOfScreen* macro returns the display of the specified screen.

The *EventMaskOfScreen* macro returns the root event mask of the root window for the specified screen at connecti setup time.

The *HeightOfScreen* macro returns the height of the specified screen.

The *HeightMMOfScreen* macro returns the height of the specified screen in millimeters.

The *MaxCmapsOfScreen* macro returns the maximum number of installed colormaps supported by the specified screen.

The *MinCmapsOfScreen* macro returns the minimum number of installed colormaps supported by the specified screen.

The *PlanesOfScreen* macro returns the number of planes in the root window of the specified screen.

The *RootWindowOfScreen* macro returns the root window of the specified screen.

The *WidthOfScreen* macro returns the width of the specified screen.

The *WidthMMOfScreen* macro returns the width of the specified screen in millimeters.

SEE ALSO

AllPlanes(3X11), ImageByteOrder(3X11), IsCursorKey(3X11)

NAME

ImageByteOrder, BitmapBitOrder, BitmapPad, BitmapUnit, DisplayHeight, DisplayHeightMM, DisplayWidth, DisplayWidthMM - image format macros

SYNOPSIS

ImageByteOrder(display)

BitmapBitOrder(display)

BitmapPad(display)

BitmapUnit(display)

DisplayHeight(display, screen_number)

DisplayHeightMM(display, screen_number)

DisplayWidth(display, screen_number)

DisplayWidthMM(display, screen_number)

ARGUMENTS

display	Specifies the connection to the X server.
screen_number	Specifies the appropriate screen number on the host server.

DESCRIPTION

The *ImageByteOrder* macro specifies the required byte order for images for each scanline unit in XY format (bitmap) or for each pixel value in Z format.

The *BitmapBitOrder* macro returns *LSBFirst* or *MSBFirst* to indicate whether the leftmost bit in the bitmap as displayed on the screen is the least or most significant bit in the unit.

The *BitmapPad* macro returns the number of bits that each scanline must be padded.

The *BitmapUnit* macro returns the size of a bitmap's scanline unit in bits.

The *DisplayHeight* macro returns the height of the specified screen in pixels.

The *DisplayHeightMM* macro returns the height of the specified screen in millimeters.

The *DisplayWidth* macro returns the width of the screen in pixels.

The *DisplayWidthMM* macro returns the width of the specified screen in millimeters.

SEE ALSO

AllPlanes(3X11), BlackPixelOfScreen(3X11), IsCursorKey(3X11)
Xlib - C Language X Interface

NAME

IsCursorKey, IsFunctionKey, IsKeypadKey, IsMiscFunctionKey, IsModiferKey, IsPFKey - keysym classification macros

SYNOPSIS

IsCursorKey(keysym)

IsFunctionKey(keysym)

IsKeypadKey(keysym)

IsMiscFunctionKey(keysym)

IsModifierKey(keysym)

IsPFKey(keysym)

ARGUMENTS

keysym Specifies the KeySym that is to be tested.

DESCRIPTION

The *IsCursorKey* macro returns *True* if the specified KeySym is a cursor key.

The *IsFunctionKey* macro returns *True* if the KeySym is a function key.

The *IsKeypadKey* macro returns *True* if the specified KeySym is a keypad key.

The *IsMiscFunctionKey* macro returns *True* if the specified KeySym is a miscellaneous function key.

The *IsModiferKey* macro returns *True* if the specified KeySym is a modifier key.

The *IsPFKey* macro returns *True* if the specified KeySym is a PF key.

SEE ALSO

AllPlanes(3X11), BlackPixelOfScreen(3X11), ImageByteOrder(3X11)

NAME

XAddHost, XAddHosts, XListHosts, XRemoveHost, XRemoveHosts, XSetAccessControl, XEnableAccessControl, XDisableAccessContro - control host access

SYNOPSIS

```
XAddHost(display, host)
      Display *display;
      XHostAddress *host;

XAddHosts(display, hosts, num_hosts)
      Display *display;
      XHostAddress *hosts;
      int num_hosts;

XHostAddress *XListHosts(display, nhosts_return, state_return)
      Display *display;
      int *nhosts_return;
      Bool *state_return;

XRemoveHost(display, host)
      Display *display;
      XHostAddress *host;

XRemoveHosts(display, hosts, num_hosts)
      Display *display;
      XHostAddress *hosts;
      int num_hosts;

XSetAccessControl(display, mode)
      Display *display;
      int mode;

XEnableAccessControl(display)
      Display *display;

XDisableAccessControl(display)
      Display *display;
```

ARGUMENTS

display	Specifies the connection to the X server.
host	Specifies the host that is to be added or removed.
hosts	Specifies each host that is to be added or removed.
mode	Specifies the mode. You can pass *EnableAccess* or *DisableAccess*
nhosts_return	Returns the number of hosts currently in the access control list.
num_hosts	Specifies the number of hosts.
state_return	Returns the state of the access control.

DESCRIPTION

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.

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.

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*.

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.

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.

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.

The *XEnableAccessControl* function enables the use of the access control list at each connection setup.

XEnableAccessControl can generate a *BadAccess* error.

The *XDisableAccessControl* function disables the use of the access control list at each connection setup.

XDisableAccessControl can generate a *BadAccess* error.

DIAGNOSTICS

BadAccess A client attempted to modify the access control list from other than the local (or otherwise authorized) host.

BadValue Some numeric value falls outside 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 can generate this error.

NAME

XAllocColor, XAllocNamedColor, XAllocColorCells, XAllocColorPlanes, XFreeColors - allocate and free colors

SYNOPSIS

```
Status XAllocColor(display, colormap, screen_in_out)
      Display *display;
      Colormap colormap;
      XColor *screen_in_out;

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;

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;

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;

XFreeColors(display, colormap, pixels, npixels, planes)
      Display *display;
      Colormap colormap;
      unsigned long pixels[];
      int npixels;
      unsigned long planes;
```

ARGUMENTS

color_name	Specifies the color name string (for example, red) whose color definition structure you want returned.
colormap	Specifies the colormap.
contig	Specifies a Boolean value that indicates whether the planes must be contiguous.
display	Specifies the connection to the X server.
exact_def_return	Returns the exact RGB values.
ncolors	Specifies the number of pixel values that are to be returned in the pixels_return array.
npixels	Specifies the number of pixels.
nplanes	Specifies the number of plane masks that are to be returned in the plane masks array.

nreds
ngreens
nblues
Specify the number of red, green, and blue planes. The value you pass must be nonnegative.

pixels Specifies an array of pixel values.

pixels_return Returns an array of pixel values.

plane_mask_return Returns an array of plane masks.

planes Specifies the planes you want to free.

rmask_return
gmask_return
bmask_return Return bit masks for the red, green, and blue planes.

screen_def_return Returns the closest RGB values provided by the hardware.

screen_in_out Specifies and returns the values actually used in the colormap.

DESCRIPTION

The *XAllocColor* function allocates a read-only colormap entry corresponding to the closest RGB values 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 values actually used. The corresponding colormap cell is read-only. In addition, *XAllocColor* returns nonzero if it succeeded or zero if it failed. Read-only colormap cells are shared among clients. 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.

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. You should use the ISO Latin-1 encoding; uppercase and lowercase do not matter.

XAllocNamedColor can generate a *BadColor* error.

The *XAllocColorCells* function allocates read/write color cells. The number of colors must be positive and the number of planes nonnegative, 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 nonzero if it succeeded or zero if it failed.

XAllocColorCells can generate *BadColor* and *BadValue* errors.

The specified ncolors must be positive; and nreds, ngreens, and nblues must be nonnegative, 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 nonzero if it succeeded or zero if it failed.

XAllocColorPlanes can generate *BadColor* and *BadValue* errors.

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 the following 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.

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), 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.

DIAGNOSTICS

BadAccess	A client attempted to free a color map entry that it did not already allocate.
BadAccess	A client attempted to store into a read-only color map entry.
BadColor	A value for a Colormap argument does not name a defined Colormap.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XCreateColormap(3X11), XQueryColor(3X11), XStoreColors(3X11)

NAME

XAllowEvents - release queued events

SYNOPSIS

```
XAllowEvents(display, event_mode, time)
      Display *display;
      int event_mode;
      Time time;
```

ARGUMENTS

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*

DESCRIPTION

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.

XAllowEvents can generate a *BadValue* error.

DIAGNOSTICS

BadValue Some numeric value falls outside 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 can generate this error.

NAME

XChangeKeyboardControl, XGetKeyboardControl, XAutoRepeatOn, XAutoRepeatOff, XBell, XQueryKeymap - manipulate keyboard settings

SYNOPSIS

```
XChangeKeyboardControl(display, value_mask, values)
      Display *display;
      unsigned long value_mask;
      XKeyboardControl *values;

XGetKeyboardControl(display, values_return)
      Display *display;
      XKeyboardState *values_return;

XAutoRepeatOn(display)
      Display *display;

XAutoRepeatOff(display)
      Display *display;

XBell(display, percent)
      Display *display;
      int percent;

XQueryKeymap(display, keys_return)
      Display *display;
      char keys_return[32];
```

ARGUMENTS

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.
percent	Specifies the volume for the bell, which can range from -100 to 100 inclusive.
value_mask	Specifies one value for each bit set to 1 in the mask.
values	Specifies which controls to change. This mask is the bitwise inclusive OR of the valid control mask bits.
values_return	Returns the current keyboard controls in the specified *XKeyboardState* structure.

DESCRIPTION

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.

The *XGetKeyboardControl* function returns the current control values for the keyboard to the *XKeyboardState* structure.

The *XAutoRepeatOn* function turns on auto-repeat for the keyboard on the specified display.

The *XAutoRepeatOff* function turns off auto-repeat for the keyboard on the specified display.

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 nonnegative 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.

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.

DIAGNOSTICS

BadMatch Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue Some numeric value falls outside 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 can generate this error.

SEE ALSO

XChangeKeyboardMapping(3X11), XSetPointerMapping(3X11)

NAME

XChangeKeyboardMapping, XGetKeyboardMapping, XDisplayKeycodes, XSetModifierMapping, XGetModifierMapping, XNewModifiermap, XInsertModifiermapEntry, XDeleteModifiermapEntry, XFreeModifierMap - manipulate keyboard encoding

SYNOPSIS

```
XChangeKeyboardMapping(display, first_keycode, keysyms_per_keycode, keysyms, num_codes)
      Display *display;
      int first_keycode;
      int keysyms_per_keycode;
      KeySym *keysyms;
      int num_codes;

KeySym *XGetKeyboardMapping(display, first_keycode, keycode_count,
                            keysyms_per_keycode_return)
      Display *display;
      KeyCode first_keycode;
      int keycode_count;
      int *keysyms_per_keycode_return;

XDisplayKeycodes(display, min_keycodes_return, max_keycodes_return)
        Display *display;
        int *min_keycodes_return, max_keycodes_return;

int XSetModifierMapping(display, modmap)
        Display *display;
        XModifierKeymap *modmap;

XModifierKeymap *XGetModifierMapping(display)
      Display *display;


XModifierKeymap *XNewModifiermap(max_keys_per_mod)
        int max_keys_per_mod;

XModifierKeymap *XInsertModifiermapEntry(modmap, keycode_entry, modifier)
     XModifierKeymap *modmap;
     KeyCode keycode_entry;
     int modifier;

XModifierKeymap *XDeleteModifiermapEntry(modmap, keycode_entry, modifier)
     XModifierKeymap *modmap;
     KeyCode keycode_entry;
     int modifier;

XFreeModifiermap(modmap)
        XModifierKeymap *modmap;
```

ARGUMENTS

display Specifies the connection to the X server.

first_keycode Specifies the first KeyCode that is to be changed or returned.

keycode_count Specifies the number of KeyCodes that are to be returned.

keycode_entry Specifies the KeyCode.

keysyms Specifies a pointer to an array of KeySyms.

keysyms_per_keycode
Specifies the number of KeySyms per KeyCode.

keysyms_per_keycode_return
Returns the number of KeySyms per KeyCode.

max_keys_per_mod Specifies the number of KeyCode entries preallocated to the modifiers in the map.

max_keycodes_return
Returns the maximum number of KeyCodes.

min_keycodes_return Returns the minimum number of KeyCodes.

modifier Specifies the modifier.

modmap Specifies a pointer to the *XModifierKeymap* structure.

num_codes Specifies the number of KeyCodes that are to be changed.

DESCRIPTION

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 *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.

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.

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 nonzero KeyCodes have meaning in each set, and zero KeyCodes are ignored. In addition, all of the nonzero KeyCodes must be in the range specified by min_keycode and max_keycode in the *Display* structure, or a *BadValue* error results. No KeyCode may appear twice in the entire map, 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.

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.

The *XNewModifiermap* function returns a pointer to *XModifierKeymap* structure for later use.

The *XInsertModifiermapEntry* function adds the specified KeyCode to the set that controls the specified modifier and returns the resulting *XModifierKeymap* structure (expanded as needed).

The *XDeleteModifiermapEntry* function deletes the specified KeyCode from the set that controls the specified modifier and returns a pointer to the resulting *XModifierKeymap* structure.

The *XFreeModifiermap* function frees the specified *XModifierKeymap* structure.

DIAGNOSTICS

BadAlloc — The server failed to allocate the requested resource or server memory.

BadValue — Some numeric value falls outside 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 can generate this error.

SEE ALSO

XSetPointerMapping(3X11)

NAME

XChangePointerControl, XGetPointerControl - control pointer

SYNOPSIS

```
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;

XGetPointerControl(display, accel_numerator_return, accel_denominator_return,
                   threshold_return)
    Display *display;
    int *accel_numerator_return, *accel_denominator_return;
    int *threshold_return;
```

ARGUMENTS

accel_denominator Specifies the denominator for the acceleration multiplier.

accel_denominator_return
Returns the denominator for the acceleration multiplier.

accel_numerator Specifies the numerator for the acceleration multiplier.

accel_numerator_return
Returns the numerator for the acceleration multiplier.

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.

threshold Specifies the acceleration threshold.

threshold_return Returns the acceleration threshold.

DESCRIPTION

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.

The *XGetPointerControl* function returns the pointer's current acceleration multiplier and acceleration threshold.

DIAGNOSTICS

BadValue Some numeric value falls outside 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 can generate this error.

NAME

XChangeSaveSet, XAddToSaveSet, XRemoveFromSaveSet - change a client's save set

SYNOPSIS

```
XChangeSaveSet (display, w, change_mode)
      Display *display;
      Window w;
      int change_mode;

XAddToSaveSet (display, w)
      Display *display;
      Window w;

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

ARGUMENTS

change_mode	Specifies the mode. You can pass *SetModeInsert* or *SetModeDelete*
display	Specifies the connection to the X server.
w	Specifies the window that you want to add or delete from the client's save-set.

DESCRIPTION

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.

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.

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.

DIAGNOSTICS

BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadValue	Some numeric value falls outside 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 can generate this error.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XReparentWindow(3X11)

NAME

XChangeWindowAttributes, XSetWindowBackground, XSetWindowBackgroundPixmap, XSetWindowBorder, XSetWindowBorderPixmap - change window attributes

SYNOPSIS

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

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

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

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

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

ARGUMENTS

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.
background_pixel	Specifies the pixel that is to be used for the background.
background_pixmap	Specifies the background pixmap, *ParentRelative*, or *None*.
border_pixel	Specifies the entry in the colormap.
border_pixmap	Specifies the border pixmap or ICopyFromParent.
display	Specifies the connection to the X server.
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.
w	Specifies the window.

DESCRIPTION

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,

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.

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.

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.

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.

DIAGNOSTICS

BadAccess	A client attempted to free a color map entry that it did not already allocate.
BadAccess	A client attempted to store into a read-only color map entry.
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.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadMatch	An *InputOnly* window locks this attribute.
BadPixmap	A value for a Pixmap argument does not name a defined Pixmap.
BadValue	Some numeric value falls outside 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 can generate this error.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XConfigureWindow(3X11), XCreateWindow(3X11), XDestroyWindow(3X11), XMapWindow(3X11), XRaiseWindow(3X11), XUnmapWindow(3X11)

NAME

XClearArea, XClearWindow - clear area or window

SYNOPSIS

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

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

ARGUMENTS

display	Specifies the connection to the X server.
exposures	Specifies a Boolean value that indicates if *Expose* events are to be generated.
w	Specifies the window.
width *height*	Specify the width and height, which are the dimensions of the rectangle.
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.

DESCRIPTION

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 indow 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.

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.

DIAGNOSTICS

BadMatch	An *InputOnly* window is used as a Drawable.
BadValue	Some numeric value falls outside 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 can generate this error.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XCopyArea(3X11)

NAME

XConfigureWindow, XMoveWindow, XResizeWindow, XMoveResizeWindow, XSetWindowBorderWidth - configure windows

SYNOPSIS

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

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

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

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

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

ARGUMENTS

display	Specifies the connection to the X 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 a pointer to the *XWindowChanges* structure.
w	Specifies the window to be reconfigured, moved, or resized..
width	Specifies the width of the window border.
width *height*	Specify the width and height, which are the interior dimensions of the window.
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 or define the new position of the window relative to its parent.

DESCRIPTION

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.

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.

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.

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.

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

XSetWindowBorderWidth can generate a *BadWindow* error.

DIAGNOSTICS

BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadValue	Some numeric value falls outside 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 can generate this error.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(3X11), XCreateWindow(3X11), XDestroyWindow(3X11), XMapWindow(3X11), XRaiseWindow(3X11), XUnmapWindow(3X11)

NAME

XCopyArea, XCopyPlane - copy areas

SYNOPSIS

```
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;

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;
```

ARGUMENTS

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.

display — Specifies the connection to the X server.

gc — Specifies the GC.

plane — Specifies the bit plane. You must set exactly one bit to 1.

src
dest — Specify the source and destination rectangles to be combined.

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.

DESCRIPTION

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.

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 sup n%, where *n* is the depth of scr, 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.

DIAGNOSTICS

BadDrawable — A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC — A value for a GContext argument does not name a defined GContext.

BadMatch — An *InputOnly* window is used as a Drawable.

BadMatch — Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue — Some numeric value falls outside 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 can generate this error.

SEE ALSO

XClearArea(3X11)

NAME

XCreateColormap, XCopyColormapAndFree, XFreeColormap, XSetWindowColormap - create, copy, or destroy colormaps

SYNOPSIS

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

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

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

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

ARGUMENTS

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

colormap Specifies the colormap that you want to create, copy, set, or destroy.

display Specifies the connection to the X server.

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

w Specifies the window for which you want to create or set a colormap .

DESCRIPTION

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, /fP and DirectColor. /fP 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 3.1.

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.

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.

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.

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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadColor	A value for a Colormap argument does not name a defined Colormap.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadValue	Some numeric value falls outside 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 can generate this error.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocColor(3X11), XQueryColor(3X11), XStoreColors(3X11)

NAME

XCreateFontCursor, XCreatePixmapCursor, XCreateGlyphCursor - create cursors

SYNOPSIS

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

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

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;

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;
```

ARGUMENTS

background_color	Specifies the RGB values for the background of the source.
display	Specifies the connection to the X server.
foreground_color	Specifies the RGB values for the foreground of the source.
mask	Specifies the cursor's source bits to be displayed or *None.*
mask_char	Specifies the glyph character for the mask.
mask_font	Specifies the font for the mask glyph or *None.*
shape	Specifies the shape of the cursor.
source	Specifies the shape of the source cursor.
source_char	Specifies the character glyph for the source.
source_font	Specifies the font for the source glyph.
x	
y	Specify the x and y coordinates, which indicate the hotspot relative to the source's origin.

DESCRIPTION

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*).

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

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.

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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadFont	A value for a Font or GContext argument does not name a defined Font.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadPixmap	A value for a Pixmap argument does not name a defined Pixmap.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XDefineCursor(3X11), XRecolorCursor(3X11)

NAME

XCreateGC, XCopyGC, XChangeGC, XFreeGC, XGContextFromGC - create or free graphics contexts

SYNOPSIS

```
GC XCreateGC(display, d, valuemask, values)
      Display *display;
      Drawable d;
      unsigned long valuemask;
      XGCValues *values;

XCopyGC(display, src, valuemask, dest)
      Display *display;
      GC src, dest;
      unsigned long valuemask;

XChangeGC(display, gc, valuemask, values)
      Display *display;
      GC gc;
      unsigned long valuemask;
      XGCValues *values;

XFreeGC(display, gc)
      Display *display;
      GC gc;

GContext XGContextFromGC(gc)
      GC gc;
```

ARGUMENTS

d	Specifies the drawable.
dest	Specifies the destination GC.
display	Specifies the connection to the X server.
gc	Specifies the GC.
src	Specifies the components of the source GC.
valuemask	Specifies which components in the GC are to be set, copied, or changed . This argument is the bitwise inclusive OR of one or more of the valid GC component mask bits.
values	Specifies any values as specified by the valuemask.

DESCRIPTION

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.

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.

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.

The *XFreeGC* function destroys the specified GC as well as all the associated storage.

XFreeGC can generate a *BadGC* error.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadFont	A value for a Font or GContext argument does not name a defined Font.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadPixmap	A value for a Pixmap argument does not name a defined Pixmap.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XQueryBestSize(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetState(3X11), XSetTile(3X11)

NAME

XCreateImage, XGetPixel, XPutPixel, XSubImage, XAddPixel, XDestroyImage - image utilities

SYNOPSIS

```
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;

unsigned long XGetPixel(ximage, x, y)
      XImage *ximage;
      int x;
      int y;

int XPutPixel(ximage, x, y, pixel)
      XImage *ximage;
      int x;
      int y;
      unsigned long pixel;

XImage *XSubImage(ximage, x, y, subimage_width, subimage_height)
      XImage *ximage;
      int x;
      int y;
      unsigned int subimage_width;
      unsigned int subimage_height;

XAddPixel(ximage, value)
      XImage *ximage;
      long value;

int XDestroyImage(ximage)
        XImage *ximage;
```

ARGUMENTS

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.
data	Specifies a pointer to the image data.
depth	Specifies the depth of the image.
display	Specifies the connection to the X server.
format	Specifies the format for the image. You can pass *XYBitmap, XYPixmap,* or *ZPixmap*
height	Specifies the height of the image, in pixels.
offset	Specifies the number of pixels to ignore at the beginning of the scanline.
pixel	Specifies the new pixel value.

subimage_height	Specifies the height of the new subimage, in pixels.
subimage_width	Specifies the width of the new subimage, in pixels.
value	Specifies the constant value that is to be added.
visual	Specifies a pointer to the visual.
width	Specifies the width of the image, in pixels.
ximage	Specifies a pointer to the image.
x *y*	Specify the x and y coordinates.

DESCRIPTION

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 offset to a Z format 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**>.

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.

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.

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.

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.

The *XDestroyImage* function deallocates the memory associated with the *XImage* structure.

SEE ALSO

XPutImage(3X11)

NAME

XCreatePixmap, XFreePixmap - create or destroy pixmaps

SYNOPSIS

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

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

ARGUMENTS

d	Specifies which screen the pixmap is created on.
depth	Specifies the depth of the pixmap.
display	Specifies the connection to the X server.
pixmap	Specifies the pixmap.
width *height*	Specify the width and height, which define the dimensions of the pixmap.

DESCRIPTION

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 nonzero, 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.

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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadPixmap	A value for a Pixmap argument does not name a defined Pixmap.
BadValue	Some numeric value falls outside 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 can generate this error.

NAME

XCreateRegion, XSetRegion, XDestroyRegion - create or destroy regions

SYNOPSIS

Region XCreateRegion()

```
XSetRegion(display, gc, r)
      Display *display;
      GC gc;
      Region r;
```

```
XDestroyRegion(r)
      Region r;
```

ARGUMENTS

display	Specifies the connection to the X server.
gc	Specifies the GC.
r	Specifies the region.

DESCRIPTION

The *XCreateRegion* function creates a new empty 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.

The *XDestroyRegion* function deallocates the storage associated with a specified region.

SEE ALSO

XEmptyRegion(3X11), XIntersectRegion(3X11)

NAME

XCreateWindow, XCreateSimpleWindow - create windows

SYNOPSIS

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;
```

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;
```

ARGUMENTS

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.
background	Specifies the background pixel value of the window.
border	Specifies the border pixel value of the window.
border_width	Specifies the width of the created window's border in pixels.
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.
depth	Specifies the window's depth. A depth of *CopyFromParent* means the depth is taken from the parent.
display	Specifies the connection to the X server.
parent	Specifies the parent 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.
visual	Specifies the visual type. A visual of *CopyFromParent* means the visual type is taken from the parent.
width *height*	Specify the width and height, which are the created window's inside dimensions and do not include the created window's borders.
x *y*	Specify the x and y coordinates, which are the top-left outside corner of the window's borders and are relative to the inside of the parent window's borders.

DESCRIPTION

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 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.

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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
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.
BadMatch	The values do not exist for an *InputOnly* window.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadPixmap	A value for a Pixmap argument does not name a defined Pixmap.
BadValue	Some numeric value falls outside 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 can generate this error.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(3X11), XConfigureWindow(3X11), XDestroyWindow(3X11), XMapWindow(3X11), XRaiseWindow(3X11), XUnmapWindow(3X11)

NAME

XDefineCursor, XUndefineCursor - define cursors

SYNOPSIS

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

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

ARGUMENTS

cursor	Specifies the cursor that is to be displayed or *None.*
display	Specifies the connection to the X server.
w	Specifies the window.

DESCRIPTION

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.

The *XUndefineCursor* 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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadCursor	A value for a Cursor argument does not name a defined Cursor.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XCreateFontCursor(3X11), XRecolorCursor(3X11)

NAME

XDestroyWindow, XDestroySubwindows - destroy windows

SYNOPSIS

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

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

ARGUMENTS

display Specifies the connection to the X server.

w Specifies the window.

DESCRIPTION

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.

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.

DIAGNOSTICS

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(3X11), XConfigureWindow(3X11), XCreateWindow(3X11), XMapWindow(3X11), XRaiseWindow(3X11), XUnmapWindow(3X11)

NAME

XDrawArc, XDrawArcs - draw arcs

SYNOPSIS

```
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;

XDrawArcs(display, d, gc, arcs, narcs)
      Display *display;
      Drawable d;
      GC gc;
      XArc *arcs;
      int narcs;
```

ARGUMENTS

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.

arcs Specifies a pointer to an array of arcs.

d Specifies the drawable.

display Specifies the connection to the X server.

gc Specifies the GC.

narcs Specifies the number of arcs in the array.

width
height Specify the width and height, which are the major and minor axes of the arc.

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.

DESCRIPTION

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 $[x, y, width, height, angle1, angle2]$, 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 $[0, 2\pi]$ 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 $[\frac{3\pi}{2}, 2\pi]$

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.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

SEE ALSO

XDrawLine(3X11), XDrawPoint(3X11), XDrawRectangle(3X11)

NAME

XDrawImageString, XDrawImageString16 - draw image text

SYNOPSIS

```
XDrawImageString(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      char *string;
      int length;

XDrawImageString16(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XChar2b *string;
      int length;
```

ARGUMENTS

d	Specifies the drawable.
display	Specifies the connection to the X server.
gc	Specifies the GC.
length	Specifies the number of characters in the string argument.
string	Specifies the character string.
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.

DESCRIPTION

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.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

SEE ALSO

XDrawString(3X11), XDrawText(3X11)

NAME

XDrawLine, XDrawLines, XDrawSegments - draw lines and polygons

SYNOPSIS

```
XDrawLine(display, d, gc, x1, y1, x2, y2)
      Display *display;
      Drawable d;
      GC gc;
      int x1, y1, x2, y2;

XDrawLines(display, d, gc, points, npoints, mode)
      Display *display;
      Drawable d;
      GC gc;
      XPoint *points;
      int npoints;
      int mode;

XDrawSegments(display, d, gc, segments, nsegments)
      Display *display;
      Drawable d;
      GC gc;
      XSegment *segments;
      int nsegments;
```

ARGUMENTS

d	Specifies the drawable.
display	Specifies the connection to the X server.
gc	Specifies the GC.
mode	Specifies the coordinate mode. You can pass *CoordModeOrigin* or *CoordModePrevious.*
npoints	Specifies the number of points in the array.
nsegments	Specifies the number of segments in the array.
points	Specifies a pointer to an array of points.
segments	Specifies a pointer to an array of segments.
x1 *y1* *x2* *y2*	Specify the points (x1, y1) and (x2, y2) to be connected.

DESCRIPTION

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* can also generate a *BadValue* error.

DIAGNOSTICS

BadDrawable — A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC — A value for a GContext argument does not name a defined GContext.

BadMatch — An *InputOnly* window is used as a Drawable.

BadMatch — Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue — Some numeric value falls outside 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 can generate this error.

SEE ALSO

XDrawArc(3X11), XDrawPoint(3X11), XDrawRectangle(3X11)

NAME

XDrawPoint, XDrawPoints - draw points

SYNOPSIS

```
XDrawPoint(display, d, gc, x, y)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;

XDrawPoints(display, d, gc, points, npoints, mode)
      Display *display;
      Drawable d;
      GC gc;
      XPoint *points;
      int npoints;
      int mode;
```

ARGUMENTS

d	Specifies the drawable.
display	Specifies the connection to the X server.
gc	Specifies the GC.
mode	Specifies the coordinate mode. You can pass *CoordModeOrigin* or *CoordModePrevious.*
npoints	Specifies the number of points in the array.
points	Specifies a pointer to an array of points.
x	
y	Specify the x and y coordinates where you want the point drawn.

DESCRIPTION

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. *XDrawPoint* can generate *BadDrawable, BadGC, BadMatch,* and *BadValue* errors.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XDrawArc(3X11), XDrawLine(3X11), XDrawRectangle(3X11)

NAME

XDrawRectangle, XDrawRectangles - draw rectangles

SYNOPSIS

```
XDrawRectangle(display, d, gc, x, y, width, height)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      unsigned int width, height;

XDrawRectangles(display, d, gc, rectangles, nrectangles)
      Display *display;
      Drawable d;
      GC gc;
      XRectangle rectangles[];
      int nrectangles;
```

ARGUMENTS

d	Specifies the drawable.
display	Specifies the connection to the X server.
gc	Specifies the GC.
nrectangles	Specifies the number of rectangles in the array.
rectangles	Specifies a pointer to an array of rectangles.
width *height*	Specify the width and height, which specify the dimensions of the rectangle.
x *y*	Specify the x and y coordinates, which specify the upper-left corner of the rectangle.

DESCRIPTION

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, 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.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

SEE ALSO

XDrawArc(3X11), XDrawLine(3X11), XDrawPoint(3X11)

NAME

XDrawString, XDrawString16 - draw text characters

SYNOPSIS

```
XDrawString(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      char *string;
      int length;

XDrawString16(display, d, gc, x, y, string, length)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XChar2b *string;
      int length;
```

ARGUMENTS

d	Specifies the drawable.
display	Specifies the connection to the X server.
gc	Specifies the GC.
length	Specifies the number of characters in the string argument.
string	Specifies the character string.
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.

DESCRIPTION

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.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

SEE ALSO

XDrawImageString(3X11), XDrawText(3X11)

NAME

XDrawText, XDrawText16 - draw polytext text

SYNOPSIS

```
XDrawText(display, d, gc, x, y, items, nitems)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XTextItem *items;
      int nitems;

XDrawText16(display, d, gc, x, y, items, nitems)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      XTextItem16 *items;
      int nitems;
```

ARGUMENTS

d	Specifies the drawable.
display	Specifies the connection to the X server.
gc	Specifies the GC.
items	Specifies a pointer to an array of text items.
nitems	Specifies the number of text items in the array.
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.

DESCRIPTION

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.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadFont	A value for a Font or GContext argument does not name a defined Font.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.

SEE ALSO

XDrawImageString(3X11), XDrawString(3X11)

NAME

XEmptyRegion, XEqualRegion, XPointInRegion, XRectInRegion - determine if regions are empty or equal

SYNOPSIS

```
Bool XEmptyRegion(r)
      Region r;

Bool XEqualRegion(r1, r2)
      Region r1, r2;

Bool XPointInRegion(r, x, y)
      Region r;
      int x, y;

int XRectInRegion(r, x, y, width, height)
      Region r;
      int x, y;
      unsigned int width, height;
```

ARGUMENTS

r	Specifies the region.
r1 *r2*	Specify the two regions.
width *height*	Specify the width and height, which define the rectangle.
x *y*	Specify the x and y coordinates, which define the point or the coordinates of the upper-left corner of the rectangle.

DESCRIPTION

The *XEmptyRegion* function returns *True* if the region is empty.

The *XEqualRegion* function returns *True* if the two regions have the same offset, size, and shape.

The *XPointInRegion* function returns *True* if the point (x, y) is contained in the region r.

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.

SEE ALSO

XCreateRegion(3X11), XIntersectRegion(3X11)

NAME

XFillRectangle, XFillRectangles, XFillPolygon, XFillArc, XFillArcs - fill rectangles, polygons, or arcs

SYNOPSIS

```
XFillRectangle(display, d, gc, x, y, width, height)
      Display *display;
      Drawable d;
      GC gc;
      int x, y;
      unsigned int width, height;

XFillRectangles(display, d, gc, rectangles, nrectangles)
      Display *display;
      Drawable d;
      GC gc;
      XRectangle *rectangles;
      int nrectangles;

XFillPolygon(display, d, gc, points, npoints, shape, mode)
      Display *display;
      Drawable d;
      GC gc;
      XPoint *points;
      int npoints;
      int shape;
      int mode;

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;

XFillArcs(display, d, gc, arcs, narcs)
      Display *display;
      Drawable d;
      GC gc;
      XArc *arcs;
      int narcs;
```

ARGUMENTS

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.
arcs	Specifies a pointer to an array of arcs.
d	Specifies the drawable.
display	Specifies the connection to the X server.
gc	Specifies the GC.
mode	Specifies the coordinate mode. You can pass *CoordModeOrigin* or *CoordModePrevious*.
narcs	Specifies the number of arcs in the array.
npoints	Specifies the number of points in the array.

nrectangles	Specifies the number of rectangles in the array.
points	Specifies a pointer to an array of points.
rectangles	Specifies a pointer to an array of rectangles.
shape	Specifies a shape that helps the server to improve performance. You can pass Complex, Convex, or Nonconvex.
width *height*	Specify the width and height, which are the dimensions of the rectangle to be filled or the major and minor axes of the arc.
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.

DESCRIPTION

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.

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.
- If shape is *Convex,* the path is wholly convex. 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 behavior 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.

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.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	An *InputOnly* window is used as a Drawable.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XDrawArc(3X11), XDrawRectangle(3X11)

NAME

XFlush, XSync, XEventsQueued, XPending - handle output buffer or event queue

SYNOPSIS

```
XFlush(display)
      Display *display;

XSync(display, discard)
      Display *display;
      Bool discard;

int XEventsQueued(display, mode)
     Display *display;
     int mode;

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

ARGUMENTS

discard	Specifies a Boolean value that indicates whether *XSync* discards all events on the event queue.
display	Specifies the connection to the X server.
mode	Specifies the mode. You can pass *QueuedAlready, QueuedAfterFlush,* or *QueuedAfterReading.*

DESCRIPTION

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.

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 error event received by Xlib, *XSync* calls the client application's error handling routine (see section 8.12.2). 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.*

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.

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.

SEE ALSO

XIfEvent(3X11), XNextEvent(3X11), XPutBackEvent(3X11)

NAME

XFree, XNoOp - free client data

SYNOPSIS

XFree(data)
char *data;

XNoOp(display)
Display *display;

ARGUMENTS

display	Specifies the connection to the X server.
data	Specifies a pointer to the data that is to be freed.

DESCRIPTION

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.

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

NAME

XGetDefault, XResourceManagerString - get X program defaults

SYNOPSIS

```
char *XGetDefault(display, program, option)
      Display *display;
      char *program;
      char *option;

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

ARGUMENTS

display	Specifies the connection to the X server.
option	Specifies the option name.
program	Specifies the program name for the Xlib defaults (usually argv[0] of the main program).

DESCRIPTION

The *XGetDefault* function returns the value NULL if the option name specified in this argument does not exist for the program. The strings returned by *XGetDefault* are owned by Xlib and should not be modified or freed by the client.

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*.

SEE ALSO

XrmGetSearchList(3X11)

NAME

XGetVisualInfo, XMatchVisualInfo, XVisualIDFromVisual - obtain visual information

SYNOPSIS

```
XVisualInfo *XGetVisualInfo(display, vinfo_mask, vinfo_template, nitems_return)
      Display *display;
      long vinfo_mask;
      XVisualInfo *vinfo_template;
      int *nitems_return;

Status XMatchVisualInfo(display, screen, depth, class, vinfo_return)
      Display *display;
      int screen;
      int depth;
      int class;
      XVisualInfo *vinfo_return;

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

ARGUMENTS

class	Specifies the class of the screen.
depth	Specifies the depth of the screen.
display	Specifies the connection to the X server.
nitems_return	Returns the number of matching visual structures.
screen	Specifies the screen.
visual	Specifies the visual type.
vinfo_mask	Specifies the visual mask value.
vinfo_return	Returns the matched visual information.
vinfo_template	Specifies the visual attributes that are to be used in matching the visual structures.

DESCRIPTION

The *XGetVisualInfo* function returns a list of visual structures that match 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*.

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.

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

NAME

XGetWindowAttributes, XGetGeometry - get current window attribute or geometry

SYNOPSIS

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

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;
```

ARGUMENTS

border_width_return	Returns the border width in pixels.
d	Specifies the drawable, which can be a window or a pixmap.
depth_return	Returns the depth of the drawable (bits per pixel for the object).
display	Specifies the connection to the X server.
root_return	Returns the root window.
w	Specifies the window whose current attributes you want to obtain.
width_return *height_return*	Return the drawable's dimensions (width and height).
window_attributes_return	Returns the specified window's attributes in the *XWindowAttributes* structure.
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.

DESCRIPTION

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

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

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*.

DIAGNOSTICS

BadDrawable	A value for a Drawable argument does not name a defined Window or Pixmap.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XQueryPointer(3X11), XQueryTree(3X11)

NAME

XGetWindowProperty, XListProperties, XChangeProperty, XRotateWindowProperties, XDeleteProperty - obtain and change window properties

SYNOPSIS

```
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;

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

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;

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

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

ARGUMENTS

actual_format_return	Returns the actual format of the property.
actual_type_return	Returns the atom identifier that defines the actual type of the property.
bytes_after_return	Returns the number of bytes remaining to be read in the property if a partial read was performed.
data	Specifies the property data.
delete	Specifies a Boolean value that determines whether the property is deleted.
display	Specifies the connection to the X server.
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 a (char *) in the call to *XChangeProperty*.
long_length	Specifies the length in 32-bit multiples of the data to be retrieved.
long_offset	Specifies the offset in the specified property (in 32-bit quantities) where the data is to be retrieved.
mode	Specifies the mode of the operation. You can pass *PropModeReplace*, *PropModePrepend*, or *PropModeAppend*.
nelements	Specifies the number of elements of the specified data format.
nitems_return	Returns the actual number of 8-bit, 16-bit, or 32-bit items stored in the prop_return data.
num_prop	Specifies the length of the properties array.
num_prop_return	Returns the length of the properties array.
npositions	Specifies the rotation amount.
prop_return	Returns a pointer to the data in the specified format.
property	Specifies the property name.
properties	Specifies the array of properties that are to be rotated.
req_type	Specifies the atom identifier associated with the property type or .I AnyPropertyType.
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*.
w	Specifies the window whose property you want to obtain, change, rotate or delete.

DESCRIPTION

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.

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.

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 2.5. 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.

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 nonzero, 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.

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.

DIAGNOSTICS

BadAlloc — The server failed to allocate the requested resource or server memory.

BadAtom — A value for an Atom argument does not name a defined Atom.

BadValue — Some numeric value falls outside 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 can generate this error.

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XInternAtom(3X11)

NAME

XGrabButton, XUngrabButton - grab pointer buttons

SYNOPSIS

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;
```

XUngrabButton(display, button, modifiers, grab_window)

```
Display *display;
unsigned int button;
unsigned int modifiers;
Window grab_window;
```

ARGUMENTS

button	Specifies the pointer button that is to be grabbed or released or *AnyButton.*
confine_to	Specifies the window to confine the pointer in or *None.*
cursor	Specifies the cursor that is to be displayed or *None.*
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.
grab_window	Specifies the grab window.
keyboard_mode	Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync.*
modifiers	Specifies the set of keymasks or *AnyModifier.* The mask is the bitwise inclusive OR of the valid keymask bits.
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.
pointer_mode	Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync.fP*

DESCRIPTION

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.

The *XUngrabButton* function releases the passive button/key combination on the specified window if it was grabbed by this client. A modifier 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.

DIAGNOSTICS

BadCursor — A value for a Cursor argument does not name a defined Cursor.

BadValue — Some numeric value falls outside 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 can generate this error.

BadWindow — A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowEvents(3X11), XGrabPointer(3X11), XGrabKey(3X11), XGrabKeyboard(3X11)

NAME

XGrabKey, XUngrabKey - grab keyboard keys

SYNOPSIS

```
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;

XUngrabKey(display, keycode, modifiers, grab_window)
    Display *display;
    int keycode;
    unsigned int modifiers;
    Window grab_window;
```

ARGUMENTS

display	Specifies the connection to the X server.
grab_window	Specifies the grab window.
keyboard_mode	Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync*.
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.
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.
pointer_mode	Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync*.

DESCRIPTION

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.

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.

DIAGNOSTICS

BadAccess — A client attempted to grab a key/button combination already grabbed by another client.

BadValue — Some numeric value falls outside 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 can generate this error.

BadWindow — A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowAccess(3X11), XGrabButton(3X11), XGrabKeyboard(3X11), XGrabPointer(3X11)

NAME

XGrabKeyboard, XUngrabKeyboard - grab the keyboard

SYNOPSIS

```
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;

XUngrabKeyboard(display, time)
      Display *display;
      Time time;
```

ARGUMENTS

display	Specifies the connection to the X server.
grab_window	Specifies the grab window.
keyboard_mode	Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync.*
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.
pointer_mode	Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync.*
time	Specifies the time. You can pass either a timestamp or *CurrentTime.*

DESCRIPTION

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 IfFalse, *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.

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.

DIAGNOSTICS

BadValue Some numeric value falls outside 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 can generate this error.

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowEvents(3X11), XGrabButton(3X11), XGrabKey(3X11), XGrabPointer(3X11)

NAME

XGrabPointer, XUngrabPointer, XChangeActivePointerGrab - grab the pointer

SYNOPSIS

```
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;

XUngrabPointer(display, time)
      Display *display;
      Time time;

XChangeActivePointerGrab(display, event_mask, cursor, time)
      Display *display;
      unsigned int event_mask;
      Cursor cursor;
      Time time;
```

ARGUMENTS

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 IfNone.
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.
grab_window	Specifies the grab window.
keyboard_mode	Specifies further processing of keyboard events. You can pass *GrabModeSync* or *GrabModeAsync.*
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.
pointer_mode	Specifies further processing of pointer events. You can pass *GrabModeSync* or *GrabModeAsync.*
time	Specifies the time. You can pass either a timestamp or *CurrentTime.*

DESCRIPTION

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.

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.

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 a *BadCursor* and *BadValue* error.

DIAGNOSTICS

BadCursor	A value for a Cursor argument does not name a defined Cursor.
BadValue	Some numeric value falls outside 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 can generate this error.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowEvents(3X11), XGrabButton(3X11), XGrabKey(3X11), XGrabKeyboard(3X11)

NAME

XGrabServer, XUngrabServer - grab the server

SYNOPSIS

XGrabServer (display)
Display *display;

XUngrabServer (display)
Display *display;

ARGUMENTS

display Specifies the connection to the X server.

DESCRIPTION

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.

The *XUngrabServer* function restarts processing of requests and close downs on other connections. You should avoid grabbing the X server as much as possible.

SEE ALSO

XGrabButton(3X11), XGrabKey(3X11), XGrabKeyboard(3X11), XGrabPointer(3X11)

NAME

XHPAcknowledge - Send an Acknowledge to an extended input device.

SYNOPSIS

#include <X11/XHPlib.h>

```
XHPAcknowledge (display, deviceid, acknowledge)
        Display *display;
        XID             deviceid;
        unsigned        int acknowledge;
```

ARGUMENTS

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

acknowledge Specifies the acknowledge to be sent. Valid values are: **GENERAL_ACKNOWLEDGE, ACKNOWLEDGE_1, ACKNOWLEDGE_2, ACKNOWLEDGE_3, ACKNOWLEDGE_4, ACKNOWLEDGE_5, ACKNOWLEDGE_6, ACKNOWLEDGE_7.**

DESCRIPTION

This function sends an acknowledge to an input device. This allows a previously received prompt to be turned off.

A prompt is an audio or visual indication that the program controlling the input device is ready for input. The program may indicate that status by turning on a prompt on the appropriate input device.

Not all input devices support prompts and acknowledges. Any device that does support a particular prompt will also support the corresponding acknowledge.

To determine whether an input device supports a particular prompt and acknowledge, the **io_byte** field of the **XHPDeviceList** structure should be examined. The format of this structure is described in the documentation for the **XHPListInputDevices** function.

RETURN VALUE

none

DIAGNOSTICS

BadDevice An invalid device ID was specified.

BadValue An invalid acknowledge was specified.

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPPrompt(3x)

NAME

XHPChangeDeviceControl - Change the control attributes of an extension input device.

XHPChangeDeviceKeyMapping - Change the key mapping of an extension input device.

SYNOPSIS

```
XHPChangeDeviceControl (display, deviceid, value_mask, values)
        Display          *display;
        XID              deviceid;
        unsigned long    value_mask;
        XHPDeviceControl *values;

XHPChangeDeviceKeyMapping (display, deviceid, first_keycode,
                           keysyms_per_keycode, keysyms, num_codes)
        Display          *display;
        XID              deviceid;
        int              first_keycode;
        int              keysyms_per_keycode;
        KeySyms          *keysyms;
        int              num_codes;
```

ARGUMENTS

display	Specifies the connection to the X server.
deviceid	Specifies the ID of the device whose attributes are to be changed.

XHPChangeDeviceControl

value_mask	Specifies which attributes are to be changed. Each bit in the mask specifies one attribute of the specified device.
values	Specifies a pointer to the **XHPDeviceControl** structure containing the values to be changed.

XHPChangeDeviceKeyMapping

first_keycode	Specifies the first keycode that is to be changed.
keysyms_per_keycode	Specifies the number of keysyms per keycode.
keysyms	Specifies a pointer to an array of keysyms that are to be used.
num_codes	Specifies the number of keycodes that are to be changed.

DESCRIPTION

These functions are provided to support the use of input devices other than the X keyboard and X pointer device. They allow the control attributes and key mapping of those input devices to be changed. The specified device must have previously been opened (turned on) using the **XHPSetInputDevice** function.

XHPChangeDeviceControl

The attributes to be changed are specified in the **XHPDeviceControl** structure. They are not actually changed unless the corresponding bit is set in the value_mask parameter.

The following masks may be ORed into the value_mask:

```
#define DVKeyClickPercent    (1L<<0)
#define DVBellPercent        (1L<<1)
#define DVBellPitch          (1L<<2)
#define DVBellDuration       (1L<<3)
#define DVLed                (1L<<4)
#define DVLedMode            (1L<<5)
#define DVKey                (1L<<6)
#define DVAutoRepeatMode     (1L<<7)
#define DVAccelNum           (1L<<8)
#define DVAccelDenom         (1L<<9)
#define DVThreshold          (1L<<10)
```

The fields of the **XHPDeviceControl** structure are defined as follows:

```
typedef struct {
    int key_click_percent;
    int bell_percent;
    int bell_pitch;
    int bell_duration;
    int led;
    int led_mode;
    int key;
    int auto_repeat_mode;
    int accelNumerator;
    int accelDenominator;
    int threshold;
} XHPDeviceControl;
```

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, specified in milliseconds, of the bell, 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 an led is specified without an led_mode, a **BadMatch** error is generated

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 device 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 is generated.

XHPChangeDeviceKeyMapping

The **XHPChangeDeviceKeyMapping** function, starting with first_keycode, defines the symbols for the specified number of KeyCodes. The symbols for KeyCodes outside this range remained unchanged. The number of elements must be:

num_codes * keysyms_per_keycode

Otherwise, a **BadLength** error is generated. The specified first_keycode must be greater than or equal to min_keycode supplied at connection setup and stored in the **Display** structure. Otherwise, it generates a **BadValue** error. In addition, the following expression must be less than or equal to max_keycode as returned in the connection setup. Otherwise, a **BadValue** error is generated.

first_keycode + (num_codes / keysyms_per_keycode) - 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. Use a special KeySym value of **NoSymbol** to fill in unused elements for individual KeyCodes. **NoSymbol** may appear in nontrailing positions of the effective list for a KeyCode. **XHPChangeDeviceKeyMapping** generates a **DeviceMappingNotify** event.

There is no requirement that the X server interpret this mapping. It is merely stored for reading and writing by clients.

DIAGNOSTICS

XHPChangeDeviceControl can generate **BadDevice**, **BadMatch**, and **BadValue** errors.

XHPChangeDeviceKeyMapping can generate **BadDevice**, **BadLength**, and **BadValue** errors.

BadDevice The specified device does not exist, was not previously enabled via **XHPSetInputDevice**, or is the X system pointer or X system keyboard.

BadMatch An LED was specified but no valid LED mode, or a key was specified but no valid AutoRepeat mode.

BadValue One of the values specified was beyond the range of valid values.

BadLength The number of elements passed was not equal to keysyms_per_code times num_codes.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPGetDeviceKeyMapping(3x)
XGetKeyboardMapping(3x)
XChangeKeyboardMapping(3x)
XHPGetDeviceControl(3x)
XGetKeyboardControl(3x)
XChangeKeyboardControl(3x)
XGetPointerControl(3x)
XChangePointerControl(3x)

NAME

XHPConvertLookup - convert key event into keysym and characters

SYNOPSIS

int
XHPConvertLookup(event_struct, buffer_return, bytes_buffer, keysym_return, status_in_out, convert_routine)
XKeyEvent *event_struct;
char *buffer_return;
int bytes_buffer;
KeySym *keysym_return;
XComposeStatus *status_in_out;
int (*convert_routine)();

DESCRIPTION

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.

convert_routine Specifies the routine which will map the keysym into a character code, if appropriate. It also handles all other processing necessary for the input language (e.g. input server control for 16-bit languages) If this value is NULL, ISO-Latin1 characters will be returned.

The **XHPConvertLookup** function maps a key event to a keysym and a string. The modifier bits in the key event are used to indicate shift, lock, control and keyboard group.

Shift, lock and keyboard group modifier bits are used to initially set the keysym.

If the lock modifier has a caps_lock keysym associated with it, **XHPConvertLookup** interprets the lock modifier to perform caps_lock processing using the keysym value.

It then checks to see if that keysym has been rebound and if it has it returns the appropriate string in *buffer_return*.

The keysym and the modifier bits are then passed to the *convert_routine* along with *buffer_return*, *bytes_buffer*, and *status_in_out*. This routine will convert the keysym into a character code if appropriate and return it in the buffer handed to it. It will also handle control processing if appropriate. The *convert_routine* may use *status_in_out* to contain state information for input. See the manual page for any convert routine used to see how it is used. Also, if multiple input servers are running at the same time, they must each be maintained by separate *XComposeStatus* parameters.

The calling sequence for *convert_routine* is as follows:

(*convert_routine)(display, keysym, modifiers, buffer_return, bytes_buffer, status_in_out)
Display *display;
Keysym *keysym;
unsigned int modifiers;
char *buffer_return;
int bytes_buffer;
XComposeStatus *status_in_out;

The meanings of the parameters are as follows:

display The display from the key event

keysym A pointer to the keysym value of this key event.

modifiers The modifiers (state) of this key event.

buffer_return Returns the translated characters.

bytes_buffer Specifies the length of the buffer. No more than bytes_buffer of translation are returned.

status_in_out Specifies or returns the *XComposeStatus* structure or NULL.

convert_routine will return the number of characters in buffer_return.

RETURN VALUE

The return value is the length of the string returned in *buffer_return*.

EXAMPLES

The following example shows an application doing input in HP's Roman 8 character set.

```
XKeyEvent *event;
char buffer[80];
KeySym keysym;
XComposeStatus *status;
extern int XHPInputRoman8();
int count;

count = XHPConvertLookup (event, buffer, nbytes, &keysym, status, XHPInputRoman8);
```

The next example shows an application that supports all the default character sets for HP's Eurasian keyboards.

```
Display display;

count = XHPConvertLookup (event, buffer, nbytes, &keysym, status,
                    XHPGetEurasianCvt(display));
```

An application which wished to do input in ISO-LATIN1 would use:

```
count = XHPConvertLookup (event, buffer, nbytes, &keysym, status, 0);
```

An application could provide its own routine to map from keysym to character code. If an application had a routine, *InputISO_Latin2*() that mapped keysyms into ISO-LATIN2 characters it would be used as follows:

```
extern int InputISO_Latin2();

count = XHPConvertLookup (event, buffer, nbytes, &keysym, status, InputISO_Latin2);
```

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPInputChinese_s(3X), XHPInputChinese_t(3X), XHPInputJapanese(3X), XHPInputKorean(3X), XHPInputRoman8(3X), XHPSetKeyboardLanguage(3X),

INTERNATIONAL SUPPORT

8-bit and 16-bit character data.

NAME

XHPDeviceAutoRepeatOn - Turn autorepeat on for an extension input device.

XHPDeviceAutoRepeatOff - Turn autorepeat off for an extension input device.

SYNOPSIS

```
XHPDeviceAutoRepeatOn (display, deviceid, mode)
        Display         *display;
        XID             deviceid;
        unsigned        int mode;

XHPDeviceAutoRepeatOff (display, deviceid)
        Display         *display;
        XID             deviceid;
```

ARGUMENTS

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

mode Valid for **XHPDeviceAutoRepeatOn** only. Specifies the auto-repeat rate. Valid values are: **REPEAT_30**, which will cause repeats to take place every 1/30th second, and **REPEAT_60**, which will cause repeats to take place every 1/60th second.

DESCRIPTION

These functions are provided to support the use of input devices other than the X keyboard and X pointer device. They cannot be used to turn auto-repeat on or off for the X keyboard device. The core **XAutoRepeatOn** and **XAutoRepeatOff** functions should be used for that purpose.

XHPDeviceAutoRepeatOn turns on or changes auto-repeat for an extended input device that is attached to the specified display.

XHPDeviceAutoRepeatOff turns off autorepeat for an extended input device that is attached to the specified display.

RETURN VALUE

none

DIAGNOSTICS

Either function can return a **BadDevice** error. **XHPDeviceAutoRepeatOn** can return a **BadValue** error.

BadDevice An invalid device ID was specified.

BadValue An invalid mode was specified.

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

SEE ALSO

XAutoRepeatOn(3x)
XAutoRepeatOff(3x)

NAME

XHPDisableReset - Disable the reset key sequence.

SYNOPSIS

XHPDisableReset (display)
Display *display;

ARGUMENTS

display Specifies the connection to the X server.

DESCRIPTION

This function is intended for use by client programs such as **xsecure(1)** that provide security to X systems.

XHPDisableReset disables the key sequence that is pressed to reset the X server. This function will fail with a **BadAccess** error if some other client has already disabled the reset key sequence.

If a client program disables reset, then terminates, reset will automatically be re-enabled by the X server.

RETURN VALUE

none

DIAGNOSTICS

BadAccess Some other client has already disabled the reset key sequence.

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPEnableReset(3x)

NAME

XHPEnableReset - Enable the reset key sequence.

SYNOPSIS

```
XHPEnableReset      (display)
        Display     *display;
```

ARGUMENTS

display Specifies the connection to the X server.

DESCRIPTION

This function is intended for use by client programs such as **xsecure(1)** that provide security to X systems.

XHPEnableReset enables the key sequence that is pressed to reset the X server. The key sequence used is the one specified in the **/usr/lib/X11/X*pointerkeys** file, or the default sequence **Left_Shift - Control - Break** if that file does not exist.

This function is only valid for a client that has previously made a successful **XHPDisableReset** request. For other clients, a BadAccess XError will be returned.

DIAGNOSTICS

BadAccess This client did not previously disable the reset key sequence.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPDisableReset(3x)

NAME

XHPFileToPixmap - Transfer an image stored in a file into a pixmap.

SYNOPSIS

XHPFileToPixmap (display, pixmap, cmap, gc, src_x, src_y, dst_x, dst_y, width, height, filename)

```
Display        *display;
Pixmap         pixmap;
Colormap       cmap;
GC             gc;
int            src_x, src_y;
int            dest_x, dest_y;
unsigned int   width, height;
char           *filename;
```

ARGUMENTS

display	Specifies the connection to the X server.
pixmap	Specifies the pixmap ID. This is where the image will be placed.
cmap	Specifies colormap ID. If nonzero, the colormap is updated from colormap data contained in the image file.
gc	Specifies the graphics context.
src_x, src_y	Specifies the x and y coordinates of the upper left corner of the rectangle to be transfered from the image file.
dst_x, dst_y	Specifies the x and y coordinates within the window where the upper left corner of the image will be drawn.
width, height	Specifies the width and height of the subimage. These arguments define the dimensions of the rectangle.
filename	Specifies the file name to use. The format of the file name is operating system specific.

DESCRIPTION

The **XHPFileToPixmap** function is similar to **XHPFileToWindow** but has a *cmap* parameter to directly specify the colormap to be modified by the colormap stored in the image file. If *cmap* is zero, the colormap is not modified.

RETURN VALUE

The **XHPFileToPixmap** function returns one of the following values defined in */usr/include/X11/XHPImageIO.h*:

XHPIFSuccess	Successful completion.
XHPIFDrawableErr	Couldn't get drawable attributes or geometry.
XHPIFFileErr	Problem accessing file.
XHPIFRequestErr	Bad placement or size.
XHPIFAllocErr	Memory allocation failure.
XHPIFHeaderErr	File header version or size problem.

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPFileToWindow(3X)
XHPPixmapToFile(3X)
XHPQueryImageFile(3X)
XHPWindowToFile(3X)

NAME

XHPFileToWindow - Transfer an image stored in a file into a window.

SYNOPSIS

```
XHPFileToWindow (display, w, modify_cmap, gc, src_x, src_y, dst_x, dst_y, width, height, filename)
        Display         *display;
        Window          w;
        ind             modify_cmap;
        GC              gc;
        int             src_x, src_y;
        int             dest_x, dest_y;
        unsigned int    width, height;
        char            *filename;
```

ARGUMENTS

display	Specifies the connection to the X server.
w	Specifies the window ID. This is where the image will be placed.
modify_cmap	Specifies colormap modification. If zero, the window's colormap is unchanged; if nonzero, the window's colormap is updated from colormap data contained in the image file.
gc	Specifies the graphics context.
src_x, src_y	Specifies the x and y coordinates of the upper left corner of the rectangle to be transfered from the image file.
dst_x, dst_y	Specifies the x and y coordinates within the window where the upper left corner of the image will be drawn.
width, height	Specifies the width and height of the subimage. These arguments define the dimensions of the rectangle.
filename	Specifies the file name to use. The format of the file name is operating system specific.

DESCRIPTION

The **XHPFileToWindow** function transfers an image saved in a file in the (ad hoc) standard **xwd** (*X Window Dump*) format into a window.

The graphics context specified by the *gc* parameter is used to control image transfer details. Refer to the description of graphics context associated with **XPutImage** in the "Transferring Images Between Client and Server" section of the *Programming With Xlib* manual.

If the *gc* parameter is zero, the default graphics context for the display's default screen will be used.

RETURN VALUE

The **XHPFileToWindow** function returns one of the following values defined in */usr/include/X11/XHPImageIO.h*:

XHPIFSuccess	Successful completion.
XHPIFDrawableErr	Couldn't get drawable attributes or geometry.
XHPIFFileErr	Problem accessing file.
XHPIFRequestErr	Bad placement or size.
XHPIFAllocErr	Memory allocation failure.
XHPIFHeaderErr	File header version or size problem.

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPFileToPixmap(3X)
XHPPixmapToFile(3X)
XHPQueryImageFile(3X)
XHPWindowToFile(3X)
XPutImage(3X)

NAME

XHPFreeDeviceList - Free the input device list.

SYNOPSIS

#include <X11/XHPlib.h>

XHPFreeDeviceList (list)
XHPDeviceList *list;

ARGUMENTS

list Specifies the pointer to the **XHPDeviceList** array returned by a previous call to **XHPListInputDevices.**

DESCRIPTION

This function frees the array of **XHPDeviceList** structures allocated by **XHPListInputDevices.**

RETURN VALUE

none

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)

NAME

XHPGetCurrentDeviceMask - Get the current extension event mask.

SYNOPSIS

```
XHPGetCurrentDeviceMask (display, window, deviceid, mask_return)
        Display *display;
        Window  window;
        XID     deviceid;
        Mask    mask_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
window	Specifies the ID of the desired window.
deviceid	Specifies the ID of the desired extension input device.
mask_return	Address of a variable into which the server can return the mask.

DESCRIPTION

This function is provided to support the use of input devices other than the X keyboard and X pointer device.

XHPGetCurrentDeviceMask returns the current event selection mask for the specified extended input device and window. This is the mask that was specified by the calling client program on a previous **XHPSelectExtensionEvent** request.

This function is not valid for the X pointer device or the X keyboard device. The current event selection mask for those devices can be obtained by using the **XGetwindowAttribute(3x)** function.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XGetwindowAttribute(3x)
XHPSelectExtensionEvent(3x)
XHPGetExtEventMask(3x)

NAME

XHPGetDeviceFocus - Get the focus window ID for an extension input device.

XHPGetDeviceMotionEvents - Get the motion history buffer for a device.

XHPGetDeviceControl - Get the control attributes of an extension input device.

XHPGetDeviceKeyMapping - Get the key mapping of an extension input device.

XHPGetDeviceModifierMapping - Get the modifier mapping of an extension input device.

SYNOPSIS

```
XHPGetDeviceFocus (display, deviceid, focus_return, revert_to_return)
        Display *display;
        XID     deviceid;
        Window  *focus_return;
        int     *revert_to_return;

XHPTimeCoord *XHPGetDeviceMotionEvents (display, deviceid, w, start,
        stop, nevents_return)
        Display *display;
        XID     deviceid;
        Window  w;
        Time    start, stop;
        int     *nevents_return;

XHPGetDeviceControl (display, deviceid, values_return)
        Display        *display;
        XID            deviceid;
        XHPDeviceState *values_return;

KeySym
*XHPGetDeviceKeyMapping (display, deviceid, first_keycode_wanted,
                         keycode_count, keysyms_per_keycode_return)
        Display  *display;
        XID      deviceid;
        KeyCode  first_keycode_wanted;
        int      keycode_count;
        int      *keysyms_per_keycode_return;

XModifierKeyMap
*XHPGetDeviceModifierMapping (display, deviceid)
        Display  *display;
        XID      deviceid;
```

ARGUMENTS

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

XHPGetDeviceFocus Only

focus_return Specifies the address of a variable into which the server can return the ID of the window that contains the device focus.

revert_to_return Specifies the address of a variable into which the server can return the current revert_to status for the device.

XHPGetDeviceMotionEvents Only

window Must contain the constant ALLWINDOWS.

start Specifies the start time.

stop Specifies the stop time.

nevents_return Specifies the address of a variable into which the server will return the number of events in the motion buffer returned for this request.

XHPGetDeviceControl Only

values_return Specifies a pointer to an **XHPDeviceState** structure in which the device values will be returned.

XHPGetDeviceKeyMapping Only

first_keycode_wanted 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.

DESCRIPTION

These functions are provided to support the use of input devices other than the X keyboard device and X pointer device.

XHPGetDeviceFocus

XHPGetDeviceFocus allows a client to determine the focus for a particular extended input device. It returns the focus window id and the current focus state of the specified extended input device.

This function may not be used to determine the focus of the X keyboard device. The **XGetInputFocus** function should be used for that purpose.

XHPGetDeviceMotionEvents

This function returns all events in the device's motion history buffer that fall between the specified start and stop times inclusive. If the start time is in the future, or is later than the stop time, no events are returned.

For all currently supported input devices, the window parameter must be the constant **ALLWINDOWS**, which can be obtained by including **<X11/XHPlib.h>**.

The return type for this function is a structure defined as follows:

```
typedef struct {
        Time time;
        unsigned short *data;
} XHPTimeCoord;
```

In order to correctly interpret the data returned by this function, client programs need information about the device that generated that data. This information is reported by the **XHPListInputDevices** function.

The data field of the **XHPTimeCoord** structure is a pointer to an array of data items. Each item is of type short, and there is one data item per axis of motion reported by the device. The number of axes reported by the device can be determined from the **num_axes** field of the **HPDeviceList** structure for the device that is returned by the **XHPListInputDevices** function.

The value of the data items depends on the mode of the device, which is reported in the mode field of the **XHPDeviceList** function, and may be compared to constants defined in **<X11/XHPlib.h>**. If the mode is **ABSOLUTE**, the data items are the raw values generated by the device. These may be scaled by the client program using the maximum values that the device can generate for each axis of motion that it reports. The maximum value for each axis is reported in the **XHPaxis_info** structure pointed to by the **XHPDeviceList** structure.

If the mode is **RELATIVE**, the data items are the relative values generated by the device. The client program must choose an initial position for the device and maintain a current position by accumulating these relative values.

The client program should use **XFree** to free the data returned by this function.

This function is not valid for the X pointer device, or for devices that do not generate motion events. Invoking this function for an invalid device will result in a **BadDevice** error.

The motion history buffer for the X pointer device can be obtained by using the **XGetMotionEvents(3x)** function.

EXAMPLE

The following code fragment shows how positional data could be received from a graphics tablet via the motion buffer. It assumes that the client only is interested in the first two axes of motion.

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

/* Find the graphics tablet information via XHPListInputDevices */
/* Scale the input to a window whose origin is at winx, winy    */
/* and whose size is winw by winh.                              */

slist = XHPListInputDevices (disp, &ndevices);
for (i=0,list=slist; i<ndevices; i++,list++)
  if (list->type == TABLET)
        {
        XHPSetInputDevice (disp, list->x_id, (ON | DEVICE_EVENTS));
        tablet = list->x_id;
        ax = list->axes;
        if (list->mode == ABSOLUTE)
          {
          scalex = (float) winw / (float) (ax++)->max_val;
          scaley = (float) winh / (float) (ax++)->max_val;
          }
        else
          {
          scalex = 1;
          scaley = 1;
          }
        axes = list->num_axes;
        }
XHPFreeDeviceList (slist);


buf = XHPGetDeviceMotionEvents (disp, tablet, ALLWINDOWS,
        start, stop, &nevents);
savbuf = buff;

for (i=0; i<nevents; i++)
  {
  dp = buf->data;
  time = buf->time;
  x = winx + (*dp++ * scalex);
  y = winy + (*dp++ * scaley);

  /* now do something with the motion data. */

  buf++;
  }

XFree (savbuf);
```

XHPGetDeviceControl

The **XHPGetDeviceControl** function returns the control attributes of the device in the

XHPDeviceState structure.

The fields of the **XHPDeviceState** structure are defined as follows:

```
typedef struct {
    int key_click_percent;
    int bell_percent;
    unsigned int bell_pitch;
    unsigned int bell_duration;
    unsigned long led_mask;
    int global_auto_repeat;
    int accelNumerator;
    int accelDenominator;
    int threshold;
        char auto_repeats[32];
} XHPDeviceState;
```

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 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. The global_auto_repeat member can be set to either **AutoRepeatModeOn** or **AutoRepeatModeOff**.

XHPGetDeviceKeyMapping

The **XHPGetDeviceKeyMapping** function, starting with first_keycode, returns the symbols for the specified number of KeyCodes. The value specified in the first_keycode argument must be greater than or equal to min_keycode as returned in the **Display** structure at connection setup. Otherwise, **XHPGetDeviceKeyMapping** generates a **BadValue** error. In addition, the following expression must be less than or equal to max_keycode as returned in the **Display** structure at connection setup:

first_keycode + keycode_count - 1

If this is not the case, a **BadValue** error is generated. 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 + N

The keysyms_per_keycode_return value is chosen arbitrarily by the X server 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 **XHPGetDeviceKeyMapping**, use **XFree**.

XHPGetDeviceModifierMapping

The **XHPGetDeviceModifierMapping** function returns a newly created **XModifierKeymap** structure that contains the keys being used as modifiers for the specified device. 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.

DIAGNOSTICS

XHPGetDeviceKeyMapping can generate **BadDevice** and **BadValue** errors.

BadDevice The specified device does not exist, was not previously enabled via **XHPSetInputDevice**, or is the X system pointer or X system keyboard.

BadValue One of the values specified was beyond the range of valid values.

RETURN VALUE

XHPGetDeviceMotionEvents returns a pointer to the motion history buffer.

XHPGetDeviceKeyMapping returns a pointer to an array of KeySyms.

XHPGetDeviceModifierMapping returns an **XModifierMap** structure that contains the keys being used as modifiers for the device.

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XGetInputFocus(3x)
XHPListInputDevices(3x)
XHPSetDeviceFocus(3x)
XGetMotionEvents(3x)
XHPListInputDevices(3x)
XHPChangeDeviceControl(3x)
XGetKeyboardControl(3x)
XChangeKeyboardControl(3x)
XGetPointerControl(3x)
XChangePointerControl(3x)
XHPChangeDeviceKeyMapping(3x)
XGetKeyboardMapping(3x)
XChangeKeyboardMapping(3x)
XGetModifierMapping(3x)
XChangeModifierMapping(3x)
XHPSetDeviceModifierMapping(3x)

NAME

XHPGetEurasianCvt - return the convert routine for Eurasian keyboards

SYNOPSIS

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

PFI
XHPGetEurasianCvt(display)
Display *display;
```

DESCRIPTION

XHPGetEurasianCvt will return the convert routine required by **XHPConvertLookup** to convert keysyms to HP character codes. The *display* argument is used to identify the keymap currently associated with the *display* structure.

Note that calling **XHPGetEurasianCvt** forces all convert routines for all character sets that correspond to HP keyboards to be linked with your code. If this is not desired, this routine should not be used.

Users of this routine will also want to perform initialization of the keyboard previous to its use in **XHPConvertLookup**. A macro has been provided that will do this. This macro, **XHPInputInit,** should be called as part of the initialization of any client making use **XHPGetEurasianCvt**.

RETURN VALUE

XHPGetEurasianCvt returns a pointer to the convert routine if it succeeds; it returns zero upon failure.

EXAMPLES

The following is an extract from an application that supports all the default character sets for HP's Eurasian keyboards. The call to **XHPConvertLookup** converts a keyevent to a keysym, and then into a string of characters. The function returned by **XHPGetEurasianCvt** tells **XHPConvertLookup** into which HP character set the string is to be encoded.

```
Display *display;
XComposeStatus *status;

XHPInputInit(display, status);
 .
 .
 .
count = XHPConvertLookup (event, buffer, nbytes, &keysym, status,
                XHPGetEurasianCvt(display));
```

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X), XHPGetKeyboard_Id(3X), XHPInputChinese_s(3X), XHPInputChinese_t(3X), XHPInputJapanese(3X), XHPInputKorean(3X), XHPInputRoman8(3X), XHPSetKeyboardMapping(3X)

NAME

XHPGetExtEventMask - Get an extension event mask.

SYNOPSIS

XHPGetExtEventMask (display, event_constant, event_type, event_mask)

```
Display *display;
long    event_constant;
long    *event_type;        /* RETURN */
Mask    *event_mask;        /* RETURN */
```

ARGUMENTS

display Specifies the connection to the X server.

event_constant
Specifies the constant corresponding to the desired event.

event_type Specifies the address of a varible in which the server can return the event type of the desired event.

event_mask Specifies the address of a varible in which the server can return the event mask for the desired event.

DESCRIPTION

This function is provided to support the use of input devices other than the X pointer device and X keyboard device.

XHPGetExtEventMask is used by client programs to determine the event mask to be used in selecting extended events. The function passes a constant to the server that corresponds to the desired event. The server returns the event mask and event type for the desired event.

Valid constants that may be used by the client to request corresponding event masks and types are:

HPDeviceKeyPressreq
HPDeviceKeyReleasereq
HPDeviceButtonPressreq
HPDeviceButtonReleasereq
HPDeviceMotionNotifyreq
HPDeviceFocusInreq
HPDeviceFocusOutreq
HPProximityInreq
HPProximityOutreq
HPDeviceKeymapNotifyreq

For example, if an X system was configured with an extension key device, and a client program had determined the device ID of that device via **XHPListInputDevices**, and the client program wished to receive key presses from that device in window **win**, it would do the following:

```
#include <XHPlib.h>

        Display display;
        Windowwin;
        XID     deviceid;
        long    devicekeypresstype;
        Mask    devicekeypressmask;

        (connection to the X server)
        (determining the device id via XHPListInputDevices)


        XHPGetExtEventMask (display, HPDeviceKeyPressreq,
                &devicekeypresstype, &devicekeypressmask);
```

```
XHPSelectExtensionEvent (display, window, deviceid,
        devicekeypressmask);

XNextEvent (display, &event);

if (event.type = = devicekeypresstype)
   (process the event)
```

DIAGNOSTICS

BadEvent The constant passed was not one of the valid constants.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPSelectExtensionEvent(3x)
XHPGetCurrentDeviceMask(3x)

NAME

XHPGetServerMode - Get the mode of the specified screen.

SYNOPSIS

int
XHPGetServerMode (display, screen)
Display *display;
int screen;

ARGUMENTS

display Specifies the connection to the X server.

screen Specifies the number of the screen whose mode is requested.

DESCRIPTION

This function enables a client program to determine the mode of a screen. The mode returned is an integer that can be compared against one of the predefined modes. The following modes are defined:

XHPOVERLAY_MODE The X server is running in the overlay planes.

XHPIMAGE_MODE The X server is running in the image planes.

XHPSTACKED_SCREENS_MODE
The X server is running with the overlay and image planes on different screens.

XHPCOMBINED_MODE The X server is running in both the overlay and image planes.

These constants can be obtained by including the file **/usr/include/X11/XHPlib.h.**

If an invalid screen number is used, a -1 will be returned by this function.

DIAGNOSTICS

The return value indicates success or failure.

RETURN VALUE

This function returns the display mode if the request is succesful, and a -1 if an invalid screen id is used.

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

NAME

XHPGrabDevice - Grab an extended input device.

XHPGrabDeviceButton - Establish a passive grab on a button on an extension input device.

XHPGrabDeviceKey - Establish a passive grab on a key on an extension input device.

SYNOPSIS

```
XHPGrabDevice (display, deviceid, grab_window, pointer_mode,
        device_mode, owner_events, time)
        Display         *display;
        XID             deviceid;
        Window          grab_window;
        int             pointer_mode, device_mode;
        Bool            owner_events;
        Time            time;

XHPGrabDeviceButton (display, deviceid, button, modifiers, grab_window,
    owner_events, event_mask, pointer_mode, device_mode)
        Display         *display;
        XID             deviceid;
        unsigned int    button;
        unsigned int    modifiers;
        Window          grab_window;
        Bool            owner_events;
        unsigned int    event_mask;
        int             pointer_mode, device_mode;

XHPGrabDeviceKey (display, deviceid, keycode, modifiers, grab_window,
    owner_events, pointer_mode, device_mode)
        Display         *display;
        XID             deviceid;
        unsigned int    keycode;
        unsigned int    modifiers;
        Window          grab_window;
        Bool            owner_events;
        int             pointer_mode, device_mode;
```

ARGUMENTS

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

grab_window Specifies the ID of a window associated with the device specified above.

pointer_mode Only the constant **GrabModeAsync** is currently supported.

device_mode Only the constant **GrabModeAsync** is currently supported.

owner_events Specifies a boolean value of either **True** or **False**.

XHPGrabDevice

time Specifies the time. This may be either a timestamp expressed in milliseconds, or *CurrentTime.*

XHPGrabDeviceButton

button Specifies the code of the button that is to be grabbed. You can pass either the keycode or **AnyButton**.

event_mask Specifies which device events are to be reported to the client. They can be the bitwise inclusive OR of these device mask bits: **DeviceButtonPressMask**, **DeviceButtonReleaseMask**, **DevicePointerMotionmask**, **DeviceKeymapStateMask**.

XHPGrabDeviceKey

keycode Valid for **XHPGrabDeviceKey** only. Specifies the keycode of the key that is to be grabbed. You can pass either the keycode or **AnyKey**.

XHPGrabDeviceKey and XHPGrabDeviceButton Only

modifiers Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: **ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, Mod5Mask.**

You can also pass **AnyModifier**, which is equivalent to issuing the grab key request for all possible modifier combinations (including the combination of no modifiers).

DESCRIPTION

These functions are provided to support the use of input devices other than the X keyboard and X pointer device. They allow a client to grab an extension input device, or a button or key on such a device. The device must have previously been opened (turned on) using the **XHPSetInputDevice** function.

XHPGrabDevice

XHPGrabDevice causes an **HPDeviceFocusIn** event to be sent to the client doing the grab, and an **HPDeviceFocusOut** event to be sent to the window losing the device focus. **XHPGrabDevice** cannot be used to grab the X pointer device or the X keyboard device. The core **XGrabPointer** and **XGrabKeyboard** functions should be used for that purpose.

XHPGrabDeviceButton

The **XHPGrabDeviceButton** function establishes a passive grab on a device. Consequently, in the future,

- IF the device is not grabbed and the specified button is logically pressed when the specified modifier keys logically are down (and no other buttons or modifier keys are down),
- AND the grab window contains the device,
- AND a passive grab on the same device and button/key combination does not exist on any ancestor of the grab window,
- THEN the device is actively grabbed, as for **XHPGrabDevice**, the last-grab time is set to the time at which the button was pressed (as transmitted in the **DeviceButtonPress** event), and the **DeviceButtonPress** event is reported.

The interpretation of the remaining arguments is as for **XHPGrabDevice**. The active grab is terminated automatically when logical state of the device has all buttons released (independent of the logical state of the modifier keys).

Note that the logical state of a device (as seen by means of the X protocol) may lag the physical state if device event processing is frozen.

A modifier 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 Button of **AnyButton** is equivalent to issuing the request for all possible Buttoncodes. Otherwise, it is not required that the specified button be assigned to a physical button.

A **BadAccess** error is generated if some other client has issued a **XHPGrabDeviceButton** with the same device and button combination on the same window. When using **AnyModifier** or **AnyButton**, the request fails completely and the X server generates a **BadAccess** error and no grabs are established if there is a conflicting grab for any combination.

XHPGrabDeviceButton can generate **BadDevice, BadAccess, BadWindow,** and **BadValue** errors.

This function cannot be used to grab a button on the X pointer device. The core **XGrabButton** function should be used for that purpose.

XHPGrabDeviceKey

The **XHPGrabDeviceKey** function establishes a passive grab on a device. Consequently, in the future,

- IF the device is not grabbed and the specified key, which itself can be a modifier key, is logically pressed when the specified modifier keys logically are down (and no other keys are down),
- AND no other modifier keys logically are down,
- AND EITHER the grab window is an ancestor of (or is) the focus window OR the grab window is a descendent of the focus window and contains the pointer,
- AND a passive grab on the same device and key combination does not exist on any ancestor of the grab window,
- THEN the device is actively grabbed, as for **XHPGrabDevice**, the last-grab time is set to the time at which the key was pressed (as transmitted in the **DeviceKeyPress** event), and the **DeviceKeyPress** event is reported.

The interpretation of the remaining arguments is as for **XHPGrabDevice**. The active grab is terminated automatically when logical state of the device 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 means of the X protocol) may lag the physical state if device event processing is frozen.

A modifier 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 key of **AnyKey** is equivalent to issuing the request for all possible keycodes. Otherwise, the key must be in the range specified by min_keycode and max_keycode in the connection setup. If it is not within that range, **XHPGrabDeviceKey** generates a **BadValue** error.

A **BadAccess** error is generated if some other client has issued a **XHPGrabDeviceKey** with the same device and key combination on the same window. When using **AnyModifier** or **AnyKey**, the request fails completely and the X server generates a **BadAccess** error and no grabs are established if there is a conflicting grab for any combination.

XHPGrabDeviceKey can generate **BadDevice**, **BadAccess**, **BadWindow**, and **BadValue** errors.

This function cannot be used to grab a key on the X keyboard device. The core **XGrabKey** function should be used for that purpose.

DIAGNOSTICS

BadDevice	An invalid device ID was specified.
BadAccess	An grab combination was specified that conflicts with an existing grab.
BadWindow	An invalid window ID was specified.
BadValue	An invalid mode was specified.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPSetInputDevice(3x)
XHPUngrabDevice(3x)
XGrabKeyboard(3x)
XGrabPointer(3x)
XGrabButton(3x)

NAME

XHPInputChinese_s - map keysyms into Chinese_s characters.

SYNOPSIS

```
int
XHPInputChinese_s(display, keysym, modifiers, buffer_return, bytes_buffer, status_in_out)
Display *display;
KeySym *keysym;
unsigned int modifiers;
char *buffer_return;
int bytes_buffer;
XComposeStatus *status_in_out;
```

DESCRIPTION

display Specifies the connection to the X server.

keysym Specifies the keysym that is to be converted into a character.

modifiers Specifies the modifiers to be applied to the *keysym*.

buffer_return Returns the translated characters.

bytes_buffer Specifies the length of the buffer. No more than bytes_buffer of translation are returned.

status_in_out Specifies the *XComposeStatus* structure.

XHPInputChinese_s will convert *keysym* into an ASCII character, if appropriate. It will also handle 16-bit input using NLIO. If the value pointed to by *keysym* is used by the NLIO server, that value will be changed to **NoSymbol.** It will use *status_in_out* to keep the state information necessary to control NLIO. This structure must contain null values before this routine is first invoked, and must remain unchanged between uses.

This routine will also process the control modifier.

XHPInputChinese_s will use /usr/lib/nlio/serv/X11/xc0input as the NLIO server. NLIO input will be invoked when the right extend char key is hit, and it will be terminated when the left extend char key is hit. If the appropriate server is not running it will be started when it is first invoked.

Users of this routine may want to exec the NLIO server previous to it being started up when the invoke key is first struck. This can also be accomplished using **XHPNlioctl.**

The keys used to invoke and terminate the NLIO server can also be changed using **XHPNlioctl.**

This routine is intended to be used in conjunction with **XHPConvertLookup**

RETURN VALUE

The return value is the length of the string returned in *buffer_return*.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X), XHPNlioctl(3X)

INTERNATIONAL SUPPORT

8-bit and 16-bit character data.

NAME

XHPInputChinese_t - map keysyms into Chinese_t characters.

SYNOPSIS

int
XHPInputChinese_t(display, keysym, modifiers, buffer_return, bytes_buffer, status_in_out)
Display *display;
KeySym *keysym;
unsigned int modifiers;
char *buffer_return;
int bytes_buffer;
XComposeStatus *status_in_out;

DESCRIPTION

display	Specifies the connection to the X server.
keysym	Specifies the keysym that is to be converted into a character.
modifiers	Specifies the modifiers to be applied to the *keysym*.
buffer_return	Returns the translated characters.
bytes_buffer	Specifies the length of the buffer. No more than bytes_buffer of translation are returned.
status_in_out	Specifies the *XComposeStatus* structure.

XHPInputChinese_t will convert *keysym* into an ASCII character, if appropriate. It will also handle 16-bit input using NLIO. If the value pointed to by *keysym* is used by the NLIO server, that value will be changed to **NoSymbol.** It will use *status_in_out* to keep the state information necessary to control NLIO. This structure must contain null values before this routine is first invoked, and must remain unchanged between uses.

This routine will also process the control modifier.

XHPInputChinese_t will use /usr/lib/nlio/serv/X11/xt0input as the NLIO server. NLIO input will be invoked when the right extend char key is hit, and it will be terminated when the left extend char key is hit. If the appropriate server is not running it will be started when it is first invoked.

Users of this routine may want to exec the NLIO server previous to it being started up when the invoke key is first struck. This can also be accomplished using **XHPNlioctl.**

The keys used to invoke and terminate the NLIO server can also be changed using **XHPNlioctl.**

This routine is intended to be used in conjunction with **XHPConvertLookup**

RETURN VALUE

The return value is the length of the string returned in *buffer_return*.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X), XHPNlioctl(3X)

INTERNATIONAL SUPPORT

8-bit and 16-bit character data.

NAME

XHPInputISO7sub - map keysyms into ISO 7-bit substitution characters.

SYNOPSIS

int
XHPInputISO7sub(display, keysym, modifiers, buffer_return, bytes_buffer, status_in_out)
Display *display;
KeySym *keysym;
unsigned int modifiers;
char *buffer_return;
int bytes_buffer;
XComposeStatus *status_in_out;

DESCRIPTION

display	Specifies the connection to the X server.
keysym	Specifies the keysym that is to be converted into an ISO 7-bit subsitution character.
modifiers	Specifies the modifiers to be applied to the *keysym*.
buffer_return	Returns the translated characters.
bytes_buffer	Specifies the length of the buffer. No more than bytes_buffer of translation are returned.
status_in_out	Specifies the *XComposeStatus* structure.

XHPInputISO7sub will convert *keysym* into a ISO 7-bit substitution character, if appropriate. This routine will also process the control modifier. The return value is the length of the string returned in *buffer_return*. This routine is intended to be used in conjunction with **XHPConvertLookup**.

status_in_out is used to hold the information necessary to perform 7-bit substitution input. This structure must contain null values before this routine is first invoked, and must remain unchanged between uses.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X)

NAME

XHPInputJapanese - map keysyms into Japanese characters.

SYNOPSIS

int
XHPInputJapanese(display, keysym, modifiers, buffer_return, bytes_buffer, status_in_out)
Display *display;
KeySym *keysym;
unsigned int modifiers;
char *buffer_return;
int bytes_buffer;
XComposeStatus *status_in_out;

DESCRIPTION

display	Specifies the connection to the X server.
keysym	Specifies the keysym that is to be converted into a Kanji character.
modifiers	Specifies the modifiers to be applied to the *keysym*.
buffer_return	Returns the translated characters.
bytes_buffer	Specifies the length of the buffer. No more than bytes_buffer of translation are returned.
status_in_out	Specifies the *XComposeStatus* structure.

XHPInputJapanese will convert *keysym* into a Kanji8 character, if appropriate. It will also handle 16-bit input using NLIO. If the value pointed to by *keysym* is used by the NLIO server, that value will be changed to **NoSymbol.** It will use *status_in_out* to keep the state information necessary to control NLIO. This structure must contain null values before this routine is first invoked, and must remain unchanged between uses.

This routine will also process the control modifier.

XHPInputJapanese will use /usr/lib/nlio/serv/X11/xj0input as the NLIO server. The left extend char key will cause the state of NLIO input to be toggled between invoked and terminated. If the appropriate server is not running it will be started when it is first invoked.

Users of this routine may want to exec the NLIO server previous to it being started up when the invoke key is first struck. This can also be accomplished using **XHPNlioctl.**

The keys used to invoke and terminate the NLIO server can also be changed using **XHPNlioctl.**

This routine is intended to be used in conjunction with **XHPConvertLookup**

RETURN VALUE

The return value is the length of the string returned in *buffer_return*.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X), XHPNlioctl(3X)

INTERNATIONAL SUPPORT

8-bit and 16-bit character data.

NAME

XHPInputKorean - map keysyms into Korean characters.

SYNOPSIS

int
XHPInputKorean(display, keysym, modifiers, buffer_return, bytes_buffer, status_in_out)
Display *display;
KeySym *keysym;
unsigned int modifiers;
char *buffer_return;
int bytes_buffer;
XComposeStatus *status_in_out;

DESCRIPTION

display Specifies the connection to the X server.

keysym Specifies the keysym that is to be converted into a character.

modifiers Specifies the modifiers to be applied to the *keysym*.

buffer_return Returns the translated characters.

bytes_buffer Specifies the length of the buffer. No more than bytes_buffer of translation are returned.

status_in_out Specifies the *XComposeStatus* structure.

XHPInputKorean will convert *keysym* into an ASCII character, if appropriate. It will also handle 16-bit input using NLIO. If the value pointed to by *keysym* is used by the NLIO server, that value will be changed to **NoSymbol.** It will use *status_in_out* to keep the state information necessary to control NLIO. This structure must contain null values before this routine is first invoked, and must remain unchanged between uses.

This routine will also process the control modifier.

XHPInputKorean will use /usr/lib/nlio/serv/X11/xk0input as the NLIO server. NLIO input will be invoked when the right extend char key is hit, and it will be terminated when the left extend char key is hit. If the appropriate server is not running it will be started when it is first invoked.

Users of this routine may want to exec the NLIO server previous to it being started up when the invoke key is first struck. This can also be accomplished using **XHPNlioctl.**

The keys used to invoke and terminate the NLIO server can also be changed using **XHPNlioctl.**

This routine is intended to be used in conjunction with **XHPConvertLookup**

RETURN VALUE

The return value is the length of the string returned in *buffer_return*.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X), XHPNlioctl(3X)

INTERNATIONAL SUPPORT

8-bit and 16-bit character data.

NAME

XHPInputRoman8 - map keysyms into Roman8 characters.

SYNOPSIS

```
int
XHPInputRoman8(display, keysym, modifiers, buffer_return, bytes_buffer, status_in_out)
Display *display;
KeySym *keysym;
unsigned int modifiers;
char *buffer_return;
int bytes_buffer;
XComposeStatus *status_in_out;
```

DESCRIPTION

display	Specifies the connection to the X server.
keysym	Specifies the keysym that is to be converted into a Roman8 character.
modifiers	Specifies the modifiers to be applied to the *keysym*.
buffer_return	Returns the translated characters.
bytes_buffer	Specifies the length of the buffer. No more than bytes_buffer of translation are returned.
status_in_out	Specifies the *XComposeStatus* structure.

XHPInputRoman8 will convert *keysym* into a Roman8 character, if appropriate. It will also handle the input of muted characters. It will use *status_in_out* to hold the state information necessary to do this. This structure must contain null values before this routine is first invoked, and must remain unchanged between uses.

This routine will also process the control modifier.

This routine is intended to be used in conjunction with **XHPConvertLookup**

RETURN VALUE

The return value is the length of the string returned in *buffer_return*.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X)

NAME

XHPListInputDevices - List all available X input devices.

SYNOPSIS

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

typedef struct
    {
    unsigned int        resolution;     /* resolution in counts/ meter*/
    unsigned short      min_val;        /* min value this axis returns*/
    unsigned short      max_val;        /* max value this axis returns*/
    } XHPaxis_info;

typedef struct
    {
    XID                 x_id;           /* device X identifier        */
    char                *name;          /* device name                */
    XHPaxis_info        *axes;          /* pointer to axes array      */
    unsigned short      type;           /* device type                */
    unsigned short      min_keycode;    /* min X keycode from this dev*/
    unsigned short      max_keycode;    /* max X keycode from this dev*/
    unsigned char       hil_id;         /* device HIL identifier      */
    unsigned char       mode;           /* ABSOLUTE or RELATIVE            */
    unsigned char       num_axes;       /* # axes this device has     */
    unsigned char       num_buttons;    /* # buttons on this device   */
    unsigned char       num_keys;       /* # keys on this device      */
    unsigned char       io_byte;        /* device i/o descriptor byte */
    unsigned char       pad[8];         /* reserved for future use    */
    } XHPDeviceList;

XHPDeviceList *XHPListInputDevices (display, ndevices)
        Display *display;
        int *ndevices     /* RETURN */
```

ARGUMENTS

display Specifies the connection to the X server.

ndevices Specifies the address of a variable into which the server can return the number of input devices available to the X server.

DESCRIPTION

This function allows a client to determine which devices are available for X input and obtain information about those devices. The X pointer device and X keyboard are listed as well as any extension input devices available to the X server.

The X pointer device is listed first. The **x_id** field in the **XHPDeviceList** structure corresponding to the X pointer device contains the value **XPOINTER**. The X keyboard device is listed second. The **x_id** field in the **XHPDeviceList** structure corresponding to the X keyboard device contains the value **XKEYBOARD**.

XHPListInputDevices returns an array of **XHPDeviceList** structures, one for each device available to the X server. The number of entries in the list is returned in the **ndevices** parameter.

The device name is a null-terminated string consisting of an ordinal number describing the position of the device, an underscore, and the type of the device. The device position is determined by following the HIL cable from the computer to the device and counting how many devices of that same type there are. The device type is described below. As an example, if a computer was configured with a keyboard and two graphics tablets connected in that order, the device names would be as follows:

FIRST_KEYBOARD

FIRST_TABLET
SECOND_TABLET

Client programs may use this name to search for a particular instance of a particular device.

The following device types are defined in the file **<X11/XHPproto.h>**. This file is automatically included when you include **<X11/XHPlib.h>**.

MOUSE
TABLET
KEYBOARD
TOUCHSCREEN
TOUCHPAD
BUTTONBOX
BARCODE
ONE_KNOB
NINE_KNOB
TRACKBALL
QUADRATURE
ID_MODULE

These constants may be compared with the **type** field of the **XHPDeviceList** structure to locate a particular type of device.

The **min_keycode**, **max_keycode**, and **num_keys** fields are valid only for devices that have keys. They will otherwise be zero.

The **max_val** field of the **XHPAxis_info** structure contains a value that may be used to scale the input of an absolute pointing device such as a touchscreen or graphics tablet. For each axis of absolute pointing devices, the minimum and maximum values it can generate will be returned.

For relative pointing devices, the **min_val** and **max_val** fields will contain 0.

The **io_byte** field contains the information from the device I/O Descriptor byte. The 8 bits are interpreted as follows:

Bit 7 — Set if the device implements the general purpose Prompt and Acknowledge functions.

Bits 6, 5, and 4 — Indicates specific Prompt/Acknowledges implemented in the device. Zeros indicate that none of the specific Prompt/Acknowledges are implemented. A non-zero value means that Prompt/Acknowledges 1 through that value inclusive are implemented in the device.

Bit 3 — Set if the device reports Proximity In/Out information.

Bits 2, 1, and 0 — Indicates which buttons the device reports. Zeros indicate that no buttons are reported. A non-zero value means that buttons 1 through that value are reported by the device.

This function returns NULL if there are no input devices to list.

RETURN VALUE

XHPListInputDevices returns an array of XHPDeviceList structures. **XHPListInputDevices** returns NULL if no input devices are available to the X server.

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPFreeDeviceList(3x)

NAME

XHPNlioctl - configure the 16-bit input environment

SYNOPSIS

#include <X11/XHPlib.h>

```
Status XHPNlioctl(display, status_in_out, command, arg)
    Display *display;
    XComposeStatus *status_in_out;
    int command;
    char *arg;
```

DESCRIPTION

display Specifies the display

status_in_out Specifies the *XComposeStatus* structure which this routine, along with **XHPConvertLookup**, will use to maintain information about this 16-bit input server.

command specifies the command associated with this call.

arg The meaning of arg is dependent upon the value of *command*.

This routine controls the environment for the 16-bit input server maintained in *status_in_out*.

The contents of *status_in_out* must be zero before its use by either **XHPConvertLookup** or **XHPNlioctl**. Also, if multiple input servers are running at the same time, they must each be maintained by separate *XComposeStatus* paramaters.

Upon successful completion, this routine returns 0. If an error has occurred, -1 is returned and errno is set to indicate the error.

The following commands are supported by this library. Other control commands may be supported by the NLIO input servers, see the documentation for the NLIO product for details.

K16_ALT_ON

If the current state of the keyboard is in the alternate character set the value of the integer pointed to by *arg* to one, else set the value of the integer pointed to by *arg* to zero.

K16_EXEC_PROC

Exec the 16-bit input server process associated with the keyboard mapping for *display*. The state information for this server will be maintained in *status_in_out*. If the server could not be started, -1 is returned and the external variable *errno* will contain the error for the last system call that **XHPNlioctl** called. The value of *arg* is ignored.

K16_GET_STATEKEYS

Get the keysyms for the keys which control state for the Asian keyboards. The keys that are returned are those which control the state of NLIO (invoke/terminate) and those which control the state of the alternate keyboard (set/unset). The current values are returned in the *K16_state* structure.

NoSymbol is returned for all values for non-Asian keyboards. The default settings for the Asian keyboards are contained in the following table.

Japanese	
set_alternate	XK_Meta_R
unset_alternate	XK_Meta_R
invoke_nlio	XK_Meta_L
terminate_nlio	XK_Meta_L
Katakana	
set_alternate	XK_Meta_R
unset_alternate	XK_Meta_L
invoke_nlio	NoSymbol
terminate_nlio	NoSymbol
Korean, S_Chinese, T_Chinese	
set_alternate	NoSymbol
unset_alternate	NoSymbol
invoke_nlio	XK_Meta_R
terminate_nlio	XK_Meta_L

A programming example follows.

```
Display *display;
XCompose compose;
struct K16_state k16state;
KeySym invoke_nlio, terminate_nlio;
KeySym set_alternate, unset_alternate;

XHPNlioctl (display, &compose, K16_GET_STATEKEYS, &k16state);

invoke_nlio = k16state.invoke_nlio;
terminate_nlio = k16state.terminate_nlio;
set_alternate = k16state.set_nlio;
unset_alternate = k16state.unset_nlio;
```

K16_KILL_PROC

Kill the 16-bit input server process which is being maintained in *status_in_out*. No error is returned. The value of *arg* is ignored.

K16_NLIO_ON

If the 16-bit input server is currently receiving characters, set the value of the integer pointed to by *arg* to one, else set the value of the integer pointed to by *arg* to zero.

K16_SET_STATEKEYS

Set the keys which control state for the Asian keyboards. The keys that can be set are those which control the state of NLIO (invoke/terminate) and those which control the state of the alternate keyboard (set/unset). The keys are set by setting the proper flag and by specifying the keysym which controls a particular state in the *K16_state* structure.

If the keysyms that set and unset a state are the same, then that key will be a toggle key. If both keysyms are set to **NoSymbol** then that functionality is effectively disabled. Note: no checking is made for the existence of keysyms on the current keyboard. Functionality can be enabled and disabled by the use of *XChangeKeyboardMapping*.

If the current keyboard mapping for *display* is that for a non-Asian keyboard the error XHPINP_INVAL is returned. If the current keyboard is other than Japanese or Katakana and *flags* has K16_ALTSTATE set, -1 is returned and errno is set to EINVAL. If the current keyboard mapping is Katakana and *flags* has K16_NLIOSTATE set, -1 is returned and errno is set to EINVAL.

A programming example follows.

```
Display *display;
XCompose compose;
struct K16_state k16state;
```

```
KeySym invoke_nlio, terminate_nlio;
KeySym set_alternate unset_alternate

k16state.flags = K16_NLIOSTATE | K16_ALTSTATE;
k16state.invoke_nlio = invoke_nlio;
k16state.terminate_nlio = terminate_nlio;
k16state.set_alternate = set_alternate;
k16state.unset_alternate = unset_alternate;

XHPNlioctl (display, &compose, K16_SET_STATEKEYS, &k16state);
```

ERRORS

XHPNlioctl will fail if:

[EACCES]	The user is trying to exec the input server and does not have execute permission for the input server.
[EAGAIN]	The user is trying to fork the input server and a system imposed limit for the number of processes would be exceeded.
[EINVAL]	An invalid parameter was passed to the routine.
[EIO]	An error occurred in communicating with the input server.
[EMFILE]	The user is trying to start up the input server and the maximum number of file descriptors is currently open.
[ENOENT]	The user is trying to exec the input server and the file does not exist.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XGetKeyboardMapping(3X), XHPConvertLookup(3X), XHPInputChinese_s(3X), XHPInputChinese_t(3X), XHPInputJapanese(3X), XHPInputKorean(3X), XHPSetKeyboardMapping(3X)

NAME

XHPPixmapToFile - Save the contents of a rectangular pixmap area in a file.

SYNOPSIS

```
XHPPixmapToFile (display, pixmap, color_w, x, y, width, height, plane_mask, format, filename)
        Display         *display;
        Pixmap          pixmap;
        Window          color_w;
        int             x,y;
        unsigned int    width, height;
        long            plane_mask;
        int             format;
        char            *filename;
```

ARGUMENTS

display	Specifies the connection to the X server.
pixmap	Specifies the pixmap ID of the image to be saved.
color_w	Specifies a window ID. This window's colormap will be saved in the image file. Visual attributes associated with this window are used in constructing the image file header.
x, y	Specifies the x and y coordinates. These coordinates define the upper left corner of the rectangle and are relative to the origin of the drawable.
width, height	Specifies the width and height of the subimage. These arguments 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.
filename	Specifies the file name to use. The format of the file name is operating system specific.

DESCRIPTION

The **XHPPixmapToFile** function is similar to **XHPWindowToFile** but requires an additional parameter to specify the color map to be stored with the image. If the *color_w* parameter is zero, the root window associated with the pixmap is used to derive visual attributes and the colormap which get stored in the image file.

RETURN VALUE

The **XHPPixmapToFile** function returns one of the following values defined in */usr/include/X11/XHPImageIO.h*:

XHPIFSuccess	Successful completion.
XHPIFDrawableErr	Couldn't get drawable attributes or geometry.
XHPIFFileErr	Problem accessing file.
XHPIFRequestErr	Bad placement or size.
XHPIFAllocErr	Memory allocation failure.

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPFileToWindow(3X)
XHPFileToPixmap(3X)
XHPQueryImageFile(3X)
XHPWindowToFile(3x)

NAME

XHPPrompt - Send a prompt to an extended input device.

SYNOPSIS

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

XHPPrompt (display, deviceid, prompt)
        Display *display;
        XID             deviceid;
        unsigned        int prompt;
```

ARGUMENTS

display	Specifies the connection to the X server.
deviceid	Specifies the ID of the desired device.
Prompt	Specifies the Prompt to be sent. Valid values are: **GENERAL_PROMPT, PROMPT_1, PROMPT_2, PROMPT_3, PROMPT_4, PROMPT_5, PROMPT_6, PROMPT_7.**

DESCRIPTION

This function sends a prompt to an input device.

A prompt is an audio or visual indication that the program controlling the input device is ready for input. The program may indicate that status by turning on a prompt on the appropriate input device.

Not all input devices support prompts and acknowledges. Any device that does support a particular prompt will also support the corresponding acknowledge.

To determine whether an input device supports a particular prompt and acknowledge, the **io_byte** field of the **XHPDeviceList** structure should be examined. The format of this structure is described in the documentation for the **XHPListInputDevices** function.

RETURN VALUE

none

DIAGNOSTICS

BadDevice	An invalid device ID was specified.
BadValue	An invalid prompt was specified.

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPAcknowledge(3x)

NAME

XHPQueryImageFile - Return image file header structure.

SYNOPSIS

```
XHPQueryImageFile (filename, xwd_header_return)
        char             *filename;
        XWDFileHeader    *xwd_header_return;
```

ARGUMENTS

filename	Specifies the file name to use. The format of the file name is operating system specific.
xwd_header_return	Returns information about the stored image in the XWDFileHeader structure.

DESCRIPTION

The **XHPQueryImageFile** function returns an image file's header structure in the *xwd_header_return* parameter. The file */usr/include/X11/XWDFile.h* is shown in appendix E, "HP Extensions," of the *Programming With Xlib* manual.

RETURN VALUE

The **XHPQueryImageFile** function returns one of the following values defined in */usr/include/X11/XHPImageIO.h*:

XHPIFSuccess	Successful completion.
XHPIFFileErr	Problem accessing file.

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPFileToPixmap(3X)
XHPFileToWindow(3X)
XHPPixmapToFile(3X)
XHPWindowToFile(3X)

NAME

XHPSelectExtensionEvent - Select an extension event.

SYNOPSIS

```
XHPSelectExtensionEvent (display, window, deviceid, mask)
        Display *display;
        Window window;
        XID             deviceid;
        Mask            mask;
```

ARGUMENTS

display Specifies the connection to the X server.

window Specifies the window from which input is desired.

deviceid Specifies the device from which input is desired.

mask Specifies the mask of input events that are desired.

DESCRIPTION

This function is provided to support the use of input devices other than the X keyboard and X pointer device. It allows input from other input devices to be selected independently from that coming from the X pointer and keyboard.

XHPSelectExtensionEvent requests the server to send an extended event that matches the specified event mask and comes from the specified device and window. In order to use this function, the client program must first determine the appropriate *deviceid* by using the **XHPListInputDevice** function, and the appropriate event mask by using the **XHPGetExtEventMask** function. Multiple event masks returned by **XHPGetExtEventMask** may be OR'd together and specified in a single request to **XHPSelectExtensionEvent**.

This function cannot be used to select any of the core X events, or to receive input from the X Keyboard or X pointer device. The core **XSelectInput** function should be used for that purpose.

DIAGNOSTICS

BadDevice An invalid device ID was specified.

BadWindow An invalid window ID was specified.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPGetExtEventMask(3x)
XSelectInput(3x)

NAME

XHPSetDeviceFocus - Set the focus for an extended input device.

XHPSetDeviceModifierMapping - Change the modifier mapping of an extension input device.

SYNOPSIS

```
XHPSetDeviceFocus (display, deviceid, focus, revert_to, time)
        Display *display;
        XID     deviceid;
        Window  focus;
        int     revert_to;
        Time    time;

XHPSetDeviceModifierMapping (display, deviceid, modmap)
        Display          *display;
        XID              deviceid;
        XModifierKeymap  *modmap;
```

ARGUMENTS

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

XHPSetDeviceFocus Only

focus Specifies the ID of the window to which the device's focus should be set. This may be a window ID, or either *PointerRoot* or *None*.

revert_to Specifies to which window the focus of the device should revert if the focus window becomes not viewable. One of the following constants may be passed: *RevertToParent, RevertToPointerRoot,* or *RevertToNone.*

time Specifies the time. You can pass either a timestamp, expressed in milliseconds, or *CurrentTime.*

XHPSetDeviceModifierMapping Only

modmap Specifies a pointer to an XModifierKeymap structure.

DESCRIPTION

These function are provided to support the use of input devices other than the X keyboard device and X pointer device.

XHPSetDeviceFocus

XHPSetDeviceFocus allows a client to redirect the focus for a particular extended input device. This function causes an **HPDeviceFocusOut** event to be sent to the window losing the device focus, and an **HPDeviceFocusIn** event to be sent to the window gaining the device focus.

This function may not be used to set the focus of the X keyboard device. The **XSetInputFocus** function should be used for that purpose.

XHPSetDeviceModifierMapping

This function is provided to support the use of input devices other than the X keyboard and X pointer device. It allows a client program to define the keycodes that are to be used as modifiers for an extension device.

The **XHPSetDeviceModifierMapping** function specifies the KeyCodes of the keys, if any, that are to be used as modifiers for the specified input device. X permits at most eight modifier keys. If more than eight are specified in the **XModifierKeymap** structure, a **BadLength** error will be generated.

There are eight modifiers, and 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 nonzero KeyCodes have meaning in each set, and zero KeyCodes are ignored. In addition, all of the nonzero KeyCodes must be in the range specified by min_keycode and max_keycode in the **Display** structure. Otherwise, a **BadValue** error is generated. No KeyCode may appear twice in the entire map. Otherwise, a

BadValue error will be generated.

A X server can impose restrictions on how modifiers can be changed, for example, if certain keys do not generate up transitions in hardware 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, the status reply is **MappingBusy**, and none of the modifiers are changed. **XHPSetDeviceModifierMapping** generates a **MappingNotify** event when it returns **MappingSuccess**.

DIAGNOSTICS

XHPSetDeviceFocus can generate **BadMatch**, **BadWindow**, and **BadDevice** errors.

XHPSetDeviceModifierMapping can generate **BadDevice**, **BadLength**, and **BadValue** errors.

BadMatch The focus window was not viewable.

BadWindow An invalid window ID was specified.

BadDevice The specified device does not exist, was not previously enabled via **XHPSetInputDevice**, or is the X system pointer or X system keyboard.

BadLength More than 8 modifier keys were specified.

BadValue One of the values specified was beyond the range of valid values.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPSetInputDevice(3x)
XHPGetDeviceFocus(3x)
XHPGetDeviceModifierMapping(3x)
XGetModifierMapping(3x)
XSetModifierMapping(3x)

NAME

XHPSetErrorHandler - Register an X error handling routine.

SYNOPSIS

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

typedef int (*PFI) ();

PFI XHPSetErrorHandler (display, routine)
Display *display;
int (*routine) ();


int routine (display, error)
Display *display;
XErrorEvent *error;
```

DESCRIPTION

This function registers with Xlib the address of a routine to handle X errors. It is intended to be used by libraries and drivers that wish to establish an error handling routine without interfering with any error handling routine that may have been established by the client program.

XHPSetErrorHandler records one error handling routine per connection to the server. Therefore, in order for a library or driver to set up its own error handling routine without affecting that of the client, the library or driver must first have established its own connection to the server via XOpenDisplay.

When an XErrorEvent is received by the client, which error handling routine is invoked is determined by the display associated with the error. If the display matches that associated with a driver error handling routine, that error handling routine will be invoked. If it does not match any driver routine, the error handling routine established by the client, if any exists, will be invoked. Otherwise, the default Xlib error handler will be invoked.

XHPSetErrorHandler returns the address of the previously established error handler. If that error handler was the default error handler, NULL is returned.

A driver or library may remove its error handler by invoking **XHPSetErrorHandler** with a NULL error handling routine.

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

SEE ALSO

XSetErrorHandler(3x)

NAME

XHPSetInputDevice - Open a device for X input.

SYNOPSIS

#include <X11/XHPlib.h>

```
XHPSetInputDevice (display, deviceid, mode)
        Display *display;
        XID           deviceid;
        int           mode;
```

ARGUMENTS

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

mode Specifies the desired mode of access.

DESCRIPTION

This function is provided to support input devices other than the X keyboard device and the X pointer device.

Client programs use the **XHPSetInputDevice** to open an input device for extended input and to close the device. **XHPSetInputDevice** requires a mode parameter that specifies the function being requested (ON or OFF) and, if the function is ON, whether the device should be opened as an extension to the X keyboard or pointer (SYSTEM_EVENTS), or as an independently selectable device (DEVICE_EVENTS). The value of the mode parameter is set by ORing together the above constants, which may be obtained by including the file **<X11/XHPlib.h>**.

To open an input device as a device whose input can be selected independent of the X keyboard and X pointer, the client program would use the mode ON OR'd with the mode DEVICE_EVENTS. To open an input device as an extension of the X keyboard or X pointer, the client program would use the mode ON or'd with the mode SYSTEM_EVENTS. Valid values for the mode parameter are:

ON | SYSTEM_EVENTS
ON | DEVICE_EVENTS
OFF

This request will fail with a BadMode error if some other client is already using the device with a different mode.

DIAGNOSTICS

BadMode An invalid mode was specified.

BadDevice An invalid device ID was specified.

RETURN VALUE

none

FILES

/usr/include/X11/XHPlib.h

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPGetExtEventMask(3x)
XHPSelectExtensionEvent(3x)

NAME

XHPSetKeyboardMapping, XHPRefreshKeyboardMapping - set/refresh the keyboard mapping

SYNOPSIS

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

Status XHPSetKeyboardMapping(display, kbd_id, force_read)
      Display *display;
      KEYBOARD_ID kbd_id;
      int force_read;

XHPRefreshKeyboardMapping(event_map)
      XMappingEvent *event_map;

XHPSetKbdMapInit(display, kbd_id, force_read, status_in_out)
      Display *display;
      KEYBOARD_ID kbd_id;
      int force_read;
      XComposeStatus status_in_out;
```

DESCRIPTION

XHPSetKeyboardMapping allows an application to emulate other keyboards. It does this by replacing the key map associated with *display*. The keyboard to be emulated is specified by *kbd_id*.

XHPSetKeyboardMapping reads the key map from the file */usr/lib/X11/XHPKeymaps*. However, if the keyboard specified with *kbd_id* is the same as the physical keyboard recognized by the server as the input device, **XHPSetKeyboardMapping** requests the key map directly from the server. In this way, any changes to the key map (such as with **XChangeKeyboardMapping**) are preserved. This functionality can be overridden by setting *force_read* to a non-NULL value; if the value of *force_read* is non-NULL, **XHPSetKeyboardMapping** will always obtain the key map from the file */usr/lib/X11/XHPKeymaps*.

XHPSetKeyboardMapping fails if *kbd_id* is an unrecognized value or if it cannot open the key map file; the *display*'s copy of the key map is not modified.

If the server's keyboard is a non-HP keyboard, **XHPSetKeyboardMapping** returns an error code and does not modify the key map.

XHPSetKbdMapInit is a macro defined in **XHPlib.h**. It is intended for clients using **XHPGetEurasianCvt** and will perform the necessary inititialization and cleanup for that routine, as well as setting the key map for *display*.

The following values for *kbd_id* are define in <**X11/HXPlib.h**>:

KB_US_English	specifies an HP46021A US ASCII keyboard
KB_Canada_French	specifies an HP46021AC Canadian French keyboard
KB_German	specifies an HP46021AD German keyboard
KB_Euro_Spanish	specifies an HP46021AE European Spanish keyboard
KB_French	specifies an HP46021AF French keyboard
KB_Dutch	specifies an HP46021AH Dutch keyboard
KB_Katakana	specifies an HP46021AJ Katakana keyboard
KB_Canada_English	specifies an HP46021AL Canadian English keyboard
KB_Latin_Spanish	specifies an HP46021AM Latin American Spanish keyboard
KB_Norwegian	specifies an HP46021AN Norwegian keyboard
KB_Swiss_German2	specifies an HP46021AP Swiss German keyboard

KB_Swiss_German	specifies an HP46020 Swiss German keyboard
KB_Swiss_French2	specifies an HP46021AQ Swiss French keyboard
KB_Swiss_French	specifies an HP46020 Swiss French keyboard
KB_Swedish	specifies an HP46021AS Swedish keyboard
KB_UK_English	specifies an HP46021AU UK English keyboard
KB_Belgian	specifies an HP46021AW Belgian keyboard
KB_Finnish	specifies an HP46021AX Finnish keyboard
KB_Danish	specifies an HP46021AY Danish keyboard
KB_Italian	specifies an HP46021AZ Italian keyboard
KB_T_Chinese	specifies an HP46021AW#ZAA Traditional Chinese keyboard
KB_Korean	specifies an HP46021AW#ZAB Korean keyboard
KB_S_Chinese	specifies an HP46021AW#ZAC Simplified Chinese keyboard
KB_Japanese	specifies an HP46021AW#ZAL Japanese keyboard

XHPRefreshKeyboardMapping refreshes *display*'s copy of the key map and modifier information. It facilitates handling MappingNotify events when using **XHPSetKeyboardMapping** with the *force_read* argument set to NULL (i.e. when the key map for the keyboard is read from the server and not from the *XHPKeymaps* file).

If the key map has been read from *XHPKeymaps*, changes to the server's key map are irrelevant; MappingNotify events should be ignored when using **XHPSetKeyboardMapping** with *force_read* set to a non-NULL value.

RETURN VALUE

XHPSetKeyboardMapping returns zero if it succeeds, otherwise it returns one of the following values, defined in **<X11/HXPlib.h>**:

XHPKB_NOKEYFILE	The file */usr/lib/X11/XHPKeymaps* does not exist or could not be opened.
XHPKB_BADMAGIC	Either *libxHP11.a* or */usr/lib/X11/XHPKeymaps* is not the latest version.
XHPKB_BADKBID	The *kbd_id* argument is set to an improper value.
XHPKB_NONHPINPUTDEV	The keyboard attached to the server is not an HP keyboard. The key map requested was not loaded.

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPConvertLookup(3X), XHPGetEurasianCvt(3X)

NAME

XHPUngrabDevice - Release a grab of an extension input device.

XHPUngrabDeviceButton - Release a passive grab of a button on an extension input device.

XHPUngrabDeviceKey - Release a passive grab of a key on an extension input device.

SYNOPSIS

```
XHPUngrabDevice (display, deviceid, time)
        Display *display;
        XID          deviceid;
        Time         time;

XHPUngrabDeviceButton (display, deviceid, button, modifiers,
        ungrab_window)
        Display       *display;
        XID            deviceid;
        unsigned int   button;
        unsigned int   modifiers;
        Window         ungrab_window;

XHPUngrabDeviceKey (display, deviceid, keycode, modifiers,
        ungrab_window)
        Display       *display;
        XID            deviceid;
        unsigned int   keycode;
        unsigned int   modifiers;
        Window         ungrab_window;
```

ARGUMENTS

display Specifies the connection to the X server.

deviceid Specifies the ID of a previously grabbed device.

XHPUngrabDevice

time Specifies a timestamp, or *CurrentTime.*

XHPUngrabDeviceButton

button Specifies the code of the button that is to be ungrabbed. You can pass either a button or **AnyButton.**

XHPUngrabDeviceKey

keycode Specifies the keycode of the key that is to be ungrabbed. You can pass either the keycode or **AnyKey.**

XHPUngrabDeviceButton and XHPUngrabDeviceKey Only

modifiers Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: **ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, Mod5Mask.**

You can also pass AnyModifier, which is equivalent to issuing the ungrab key request for all possible modifier combinations (including the combination of no modifiers).

ungrab_window Specifies the ID of a window associated with the device specified above.

DESCRIPTION

These functions are provided to support the use of input devices other than the X keyboard and X pointer device. They allow a client to release a grab of an extended input device, or a button or key on such a device. That grab must have previously been established using the corresponding grab function.

XHPUngrabDevice

XHPUngrabDevice does not release the grab if the specified time is earlier than the last-device-grab time or is later than the current X server time. It also generates DeviceFocusIn and

DeviceFocusOut events. The X server automatically performs an **XHPUngrabDevice** if the event window for an active device grab becomes not viewable.

XHPUngrabDevice cannot be used to release a grab of the X pointer device or the X keyboard device. The core **XUngrabPointer** and **XUngrabKeyboard** functions should be used for that purpose.

XHPUngrabDeviceButton

The **XHPUngrabDeviceButton** function removes a passive grab of a button on an extension device. A modifier of **AnyModifier** is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). **XHPUngrabDeviceButton** can generate **BadDevice** and **BadWindow** errors.

XHPUngrabDeviceButton cannnot be used to ungrab a button on the X pointer device. The core **XUngrabButton** function should be used for that purpose.

XHPUngrabDeviceKey

The **XHPUngrabDeviceKey** function removes a passive grab of a key on an extension device. A modifier of **AnyModifier** is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). **XHPUngrabDeviceKey** can generate **BadDevice** and **BadWindow** errors.

XHPUngrabDeviceKey cannot be used to ungrab a key on the X keyboard device. The core **XUngrabKey** function should be used for that purpose.

DIAGNOSTICS

BadDevice An invalid device ID was specified.

BadWindow An invalid window ID was specified.

RETURN VALUE

none

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPListInputDevices(3x)
XHPSetInputDevice(3x)
XHPGrabDevice(3x)
XHPGrabDeviceButton(3x)
XHPGrabDeviceKey(3x)
XUngrabKeyboard(3x)
XUngrabPointer(3x)
XUngrabButton(3x)
XUngrabKey(3x)

NAME

XHPWindowToFile - Save the contents of a rectangular window in a file.

SYNOPSIS

```
XHPWindowToFile (display, w, x, y, width, height, plane_mask, format, filename)
        Display         *display;
        Window          w;
        int             x,y;
        unsigned int    width, height;
        long            plane_mask;
        int             format;
        char            *filename;
```

ARGUMENTS

display	Specifies the connection to the X server.
w	Specifies the window ID of the image to be saved.
x, y	Specifies the x and y coordinates. These coordinates define the upper left corner of the rectangle and are relative to the origin of the drawable.
width, height	Specifies the width and height of the subimage. These arguments 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.
filename	Specifies the file name to use. The format of the file name is operating system specific.

DESCRIPTION

XHPWindowToFile saves the specified window rectangle in the format defined by the **xwd** (*X Window Dump*) utility program. This stores a file header and a color map along with the image.

The *plane_mask* parameter controls which image planes will be included in the file. A value of ~0 (or -1) can be given to have all image planes be stored.

Images saved using **XHPWindowToFile** may be viewed using the **xwud** utility or restored under program control using **XHPFileToWindow** or **XHPFileToPixmap**.

Hardcopy of a saved image can be generated using the **xpr** utility or by translating the image into Starbase format using **xwd2sb** and piping the result to the **pcltrans** utility. This can be done under program control using the *system(3S)* library routine to issue the appropriate shell command.

RETURN VALUE

The **XHPWindowToFile** function returns one of the following values defined in */usr/include/X11/XHPImageIO.h*:

XHPIFSuccess	Successful completion.
XHPIFDrawableErr	Couldn't get drawable attributes or geometry.
XHPIFFileErr	Problem accessing file.
XHPIFRequestErr	Bad placement or size.
XHPIFAllocErr	Memory allocation failure.

FILES

none

ORIGIN

Hewlett-Packard Company

SEE ALSO

XHPFileToPixmap(3X)
XHPFileToWindow(3X)
XHPPixmapToFile(3X)

XHPQueryImageFile(3X)

NAME

XIfEvent, XCheckIfEvent, XPeekIfEvent - check the event queue with a predicate procedure

SYNOPSIS

```
XIfEvent(display, event_return, predicate, arg)
      Display *display;
      XEvent *event_return;
      Bool (*predicate)();
      char *arg;

Bool XCheckIfEvent(display, event_return, predicate, arg)
      Display *display;
      XEvent *event_return;
      Bool (*predicate)();
      char *arg;

XPeekIfEvent(display, event_return, predicate, arg)
      Display *display;
      XEvent *event_return;
      Bool (*predicate)();
      char *arg;
```

ARGUMENTS

arg	Specifies the user-supplied argument that will be passed to the predicate procedure.
display	Specifies the connection to the X server.
event_return	Returns either a copy of or 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.

DESCRIPTION

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.

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.

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.

SEE ALSO

XPutBackEvent(3X11) XNextEvent(3X11), XSendEvent(3X11)

NAME

XInstallColormap, XUninstallColormap, XListInstalledColormaps - control colormaps

SYNOPSIS

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

XUninstallColormap(display, colormap)
      Display *display;
      Colormap colormap;

Colormap *XListInstalledColormaps(display, w, num_return)
      Display *display;
      Window w;
      int *num_return;
```

ARGUMENTS

colormap	Specifies the colormap.
display	Specifies the connection to the X server.
num_return	Returns the number of currently installed colormaps.
w	Specifies the window that determines the screen.

DESCRIPTION

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.

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.

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.

DIAGNOSTICS

BadColor	A value for a Colormap argument does not name a defined Colormap.
BadWindow	A value for a Window argument does not name a defined Window.

NAME

XIntersectRegion, XUnionRegion, XUnionRectWithRegion, XSubtractRegion, XXorRegion, XOffsetRegion, XShrinkRegion - region arthmetic

SYNOPSIS

```
XIntersectRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;

XUnionRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;

XUnionRectWithRegion(rectangle, src_region, dest_region_return)
      XRectangle *rectangle;
      Region src_region;
      Region dest_region_return;

XSubtractRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;

XXorRegion(sra, srb, dr_return)
      Region sra, srb, dr_return;

XOffsetRegion(r, dx, dy)
      Region r;
      int dx, dy;

XShrinkRegion(r, dx, dy)
      Region r;
      int dx, dy;
```

ARGUMENTS

dest_region_return Returns the destination region.

dr_return Returns the result of the computation.

dx
dy Specify the x and y coordinates, which define the amount you want to the specified region.

r Specifies the region.

rectangle Specifies the rectangle.

sra
srb Specify the two regions with which you want to perform the computation.

src_region Specifies the source region to be used.

DESCRIPTION

The *XIntersectRegion* function computes the intersection of two regions.

The *XUnionRegion* function computes the union of two regions.

The *XUnionRectWithRegion* function updates the destination region from a union of the specified rectangle and the specified source region.

The *XSubtractRegion* function subtracts srb from sra and stores the results in dr_return.

The *XXorRegion* function calculates the difference between the union and intersection of two regions.

The *XOffsetRegion* function moves the specified region by a specified amount.

The *XShrinkRegion* function reduces the specified region by a specified amount. Positive values shrink the size of the region, and negative values expand the region.

SEE ALSO

XCreateRegion(3X11), XEmptyRegion(3X11)

NAME

XInternAtom, XGetAtomName - create or return atom names

SYNOPSIS

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

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

ARGUMENTS

atom	Specifies the atom for the property name you want returned.
atom_name	Specifies the name associated with the atom you want returned.
display	Specifies the connection to the X server.
only_if_exists	Specifies a Boolean value that indicates whether *XInternAtom* creates the atom.

DESCRIPTION

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.* You should use a null-terminated ISO Latin-1 string for atom_name. 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.

The *XGetAtomName* function returns the name associated with the specified atom. To free the resulting string, call *XFree*.

XGetAtomName can generate a *BadAtom* error.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadAtom	A value for an Atom argument does not name a defined Atom.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XGetWindowProperty(3X11)

NAME

XListFonts, XFreeFontNames, XListFontsWithInfo, XFreeFontInfo - obtain or free font names and information

SYNOPSIS

```
char **XListFonts(display, pattern, maxnames, actual_count_return)
      Display *display;
      char *pattern;
      int maxnames;
      int *actual_count_return;

XFreeFontNames(list)
      char *list[];

char **XListFontsWithInfo(display, pattern, maxnames, count_return, info_return)
      Display *display;
      char *pattern;
      int maxnames;
      int *count_return;
      XFontStruct **info_return;

XFreeFontInfo(names, free_info, actual_count)
      char **names;
      XFontStruct *free_info;
      int actual_count;
```

ARGUMENTS

actual_count	Specifies the actual number of matched font names returned by *XListFontsWithInfo.*
actual_count_return	Returns the actual number of font names.
count_return	Returns the actual number of matched font names.
display	Specifies the connection to the X server.
info_return	Returns a pointer to the font information.
free_info	Specifies the pointer to the font information returned by *XListFontsWithInfo.*
list	Specifies the array of strings you want to free.
maxnames	Specifies the maximum number of names to be returned.
names	Specifies the list of font names returned by *XListFontsWithInfo.*
pattern	Specifies the null-terminated pattern string that can contain wildcard characters.

DESCRIPTION

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 string should be ISO Latin-1; uppercase and lowercase do not matter. Each string is terminated by an ASCII null. 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. The client should call *XFreeFontNames* when finished with the result to free the memory.

The *XFreeFontNames* function frees the array and strings returned by *XListFonts* or *XListFontsWithInfo.*

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. To free the allocated name array, the client should call

XFreeFontNames. To free the the font information array, the client should call *XFreeFontInfo*.

The *XFreeFontInfo* function frees the the font information array.

SEE ALSO

XLoadFont(3X11), XSetFontPath(3X11)

NAME

XLoadFont, XQueryFont, XLoadQueryFont, XFreeFont, XGetFontProperty, XUnloadFont - load or unload fonts

SYNOPSIS

```
Font XLoadFont(display, name)
      Display *display;
      char *name;

XFontStruct *XQueryFont(display, font_ID)
      Display *display;
      XID font_ID;

XFontStruct *XLoadQueryFont(display, name)
      Display *display;
      char *name;

XFreeFont(display, font_struct)
      Display *display;
      XFontStruct *font_struct;

Bool XGetFontProperty(font_struct, atom, value_return)
      XFontStruct *font_struct;
      Atom atom;
      unsigned long *value_return;

XUnloadFont(display, font)
      Display *display;
      Font font;
```

ARGUMENTS

atom	Specifies the atom for the property name you want returned.
display	Specifies the connection to the X server.
font	Specifies the font.
font_ID	Specifies the font ID or the *GContext* ID.
font_struct	Specifies the storage associated with the font.
gc	Specifies the GC.
name	Specifies the name of the font, which is a null-terminated string.
value_return	Returns the value of the font property.

DESCRIPTION

The *XLoadFont* function loads the specified font and returns its associated font ID. The name should be ISO Latin-1 encoding; uppercase and lowercase do not matter. 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.

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*). To free this data, use *XFreeFontInfo.*

XLoadQueryFont can generate a *BadAlloc* error.

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 does not exist, *XLoadQueryFont* returns NULL.

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.

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.

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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadFont	A value for a Font or GContext argument does not name a defined Font.
BadName	A font or color of the specified name does not exist.

SEE ALSO

XListFonts(3X11), XSetFontPath(3X11)

NAME

XLookupKeysym, XRefreshKeyboardMapping, XLookupString, XRebindKeySym - handle keyboard input events

SYNOPSIS

```
KeySym XLookupKeysym(key_event, index)
      XKeyEvent *key_event;
      int index;

XRefreshKeyboardMapping(event_map)
      XMappingEvent *event_map;

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;

XRebindKeysym(display, keysym, list, mod_count, string, bytes_string)
      Display *display;
      KeySym keysym;
      KeySym list[];
      int mod_count;
      unsigned char *string;
      int bytes_string;
```

ARGUMENTS

buffer_return	Returns the translated characters.
bytes_buffer	Specifies the length of the buffer. No more than bytes_buffer of translation are returned.
bytes_string	Specifies the length of the string.
display	Specifies the connection to the X server.
event_map	Specifies the mapping event that is to be used.
event_struct	Specifies the key event structure to be used. You can pass *XKeyPressedEvent* or *XKeyReleasedEvent*.
index	Specifies the index into the KeySyms list for the event's KeyCode.
key_event	Specifies the *KeyPress* or *KeyRelease* event.
keysym	Specifies the KeySym that is to be .
keysym_return	Returns the KeySym computed from the event if this argument is not NULL.
list	Specifies the KeySyms to be used as modifiers.
mod_count	Specifies the number of modifiers in the modifier list.
status_in_out	Specifies or returns the *XComposeStatus* structure or NULL.
string	Specifies a pointer to the string that is copied and returned by *XLookupString*.

DESCRIPTION

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*.

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.

The *XLookupString* function is a convenience routine that maps a key event to an ISO Latin-1 string, using the modifier bits in the key event to deal with shift, lock, and control. It returns the translated string into the user's buffer. It also detects any rebound KeySyms (see *XRebindKeysym*) and returns the specified bytes. *XLookupString* returns the length of the string stored in the tag buffer. If the lock modifier has the caps lock KeySym associated with it, *XLookupString* interprets the lock modifier to perform caps lock processing.

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 *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. Note that you can rebind a KeySym that may not exist.

SEE ALSO

XStringToKeysym(3X11)

NAME

XrmMergeDatabases, XrmGetFileDatabase, XrmPutFileDatabase, XrmGetStringDatabase - manipulate resource databases

SYNOPSIS

```
void XrmMergeDatabases(source_db, target_db)
      XrmDatabase source_db, *target_db;

XrmDatabase XrmGetFileDatabase(filename)
     char *filename;

void XrmPutFileDatabase(database, stored_db)
     XrmDatabase database;
     char *stored_db;

XrmDatabase XrmGetStringDatabase(data)
     char *data;
```

ARGUMENTS

data	Specifies the database contents using a string.
database	Specifies the database that is to be used.
filename	Specifies the resource database file name.
source_db	Specifies the resource database that is to be merged into the target database.
stored_db	Specifies the file name for the stored database.
target_db	Specifies a pointer to the resource database into which the source database is to be merged.

DESCRIPTION

The *XrmMergeDatabases* function merges the contents of one database into another. It may overwrite entries in the destination database. This function is used to combine databases (for example, an application specific database of defaults and a database of user preferences). The merge is destructive; that is, the source database is destroyed.

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 lines in the format accepted by *XrmPutLineResource.* If it cannot open the specified file, *XrmGetFileDatabase* returns NULL.

The *XrmPutFileDatabase* function stores a copy of the specified database in the specified file. The file is an ASCII text file that contains lines in the format that is accepted by *XrmPutLineResource.*

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. Each line is separated by a new-line character in the format accepted by *XrmPutLineResource.*

SEE ALSO

XrmGetResource(3X11), XrmInitialize(3X11), XrmPutResource(3X11), XrmUniqueQuark(3X11)

NAME

XMapWinow, XMapRaised, XMapSubwindows - map windows

SYNOPSIS

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

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

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

ARGUMENTS

display Specifies the connection to the X server.

w Specifies the window.

DESCRIPTION

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.

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.

XMapRaised can generate a *BadWindow* error.

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.

DIAGNOSTICS

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(3X11), XConfigureWindow(3X11), XCreateWindow(3X11), XDestroyWindow(3X11), XRaiseWindow(3X11), XUnmapWindow(3X11)

NAME

NextEvent, XPeekEvent, XWindowEvent, XCheckWindowEvent, XMaskEvent, XCheckMaskEvent, XCheckTypedEvent, XCheckTypedWindowEvent - select events by type

SYNOPSIS

```
XNextEvent(display, event_return)
      Display *display;
      XEvent *event_return;

XPeekEvent(display, event_return)
      Display *display;
      XEvent *event_return;

XWindowEvent(display, w, event_mask, event_return)
      Display *display;
      Window w;
      long event_mask;
      XEvent *event_return;

Bool XCheckWindowEvent(display, w, event_mask, event_return)
      Display *display;
      Window w;
      long event_mask;
      XEvent *event_return;

XMaskEvent(display, event_mask, event_return)
      Display *display;
      long event_mask;
      XEvent *event_return;

Bool XCheckMaskEvent(display, event_mask, event_return)
      Display *display;
      long event_mask;
      XEvent *event_return;

Bool XCheckTypedEvent(display, event_type, event_return)
      Display *display;
      int event_type;
      XEvent *event_return;

Bool XCheckTypedWindowEvent(display, w, event_type, event_return)
      Display *display;
      Window w;
      int event_type;
      XEvent *event_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
event_mask	Specifies the event mask.
event_return	Returns the matched event's associated structure.
event_return	Returns the next event in the queue.
event_return	Returns a copy of the matched event's associated structure.
event_type	Specifies the event type to be compared.
w	Specifies the window whose event uou are interested in.

DESCRIPTION

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.

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.

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.

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.

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.

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.

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.

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.

SEE ALSO

XIfEvent(3X11), XPutBackEvent(3X11), XSendEvent(3X11)

NAME

XOpenDisplay, XCloseDisplay - connect or disconnect to X server

SYNOPSIS

Display *XOpenDisplay(display_name)
char *display_name;

XCloseDisplay(display)
Display *display;

ARGUMENTS

display Specifies the connection to the X server.

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

DESCRIPTION

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, UNIX domain, or DECnet communications protocols. 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 *unix* and a single colon (:) separates it from the display number, *XOpenDisplay* connects using UNIX domain IPC 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 2.2.1.

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*), 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.

NAME

XParseGeometry, XGeometry, XParseColor - parse window geometry and color

SYNOPSIS

```
int XParseGeometry(parsestring, x_return, y_return, width_return, height_return)
      char *parsestring;
      int *x_return, *y_return;
      int *width_return, *height_return;

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;

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

ARGUMENTS

bwidth	Specifies the border width.
colormap	Specifies the colormap.
position *default_position*	Specify the geometry specifications.
display	Specifies the connection to the X server.
exact_def_return	Returns the exact color value for later use and sets the *DoRed, DoGreen,* and *DoBlue* flags.
fheight *fwidth*	Specify the font height and width in pixels (increment size).
parsestring	Specifies the string you want to parse.
screen	Specifies the screen.
spec	Specifies the color name string; case is ignored.
width_return *height_return*	Return the width and height determined.
xadder *yadder*	Specify additional interior padding needed in the window.
x_return *y_return*	Return the x and y offsets.

DESCRIPTION

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*>x<*height*>][{+-}<*xoffset*>{+-}<*yoffset*>]

The items in this form 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.)

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.

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.

The *XParseColor* function provides a simple way to create a standard user interface to color. It takes a string specification of a color, typically from a command line or *XGetDefault* option, and returns the corresponding red, green, and blue values that are suitable for a subsequent call to *XAllocColor* or *XStoreColor*. The color can be specified either as a color name (for example, *XAllocNamedColor*) or as 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 (both uppercase and lowercase). When fewer than 16 bits each are specified, they represent the most-significant bits of the value. For example, #3a7 is the same as #3000a0007000. The colormap is used only to determine which screen to look up the color on. For example, you can use the screen's default colormap.

If the initial character is a sharp sign but the string otherwise fails to fit the above formats or if the initial character is not a sharp sign and the named color does not exist in the server's database, *XParseColor* fails and returns zero.

XParseColor can generate a *BadColor* error.

DIAGNOSTICS

BadColor A value for a Colormap argument does not name a defined Colormap.

NAME

XPolygonRegion, XClipBox - generate regions

SYNOPSIS

```
Region XPolygonRegion(points, n, fill_rule)
      XPoint points[];
      int n;
      int fill_rule;

XClipBox(r, rect_return)
      Region r;
      XRectangle *rect_return;
```

ARGUMENTS

fill_rule	Specifies the fill-rule you want to set for the specified GC. You can pass *EvenOddRule* or *WindingRule*.
n	Specifies the number of points in the polygon.
points	Specifies an array of points.
r	Specifies the region.
rect_return	Returns the smallest enclosing rectangle.

DESCRIPTION

The *XPolygonRegion* function returns a region for the polygon defined by the points array. For an explanation of fill_rule, see *XCreateGC*.

The *XClipBox* function returns the smallest rectangle enclosing the specified region.

NAME

XPutBackEvent - put events back on the queue

SYNOPSIS

XPutBackEvent (display, event)
Display *display;
XEvent *event;

ARGUMENTS

display	Specifies the connection to the X server.
event	Specifies a pointer to the event.

DESCRIPTION

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.*

SEE ALSO

XIfEvent(3X11), XNextEvent(3X11), XSendEvent(3X11)

NAME

XPutImage, XGetImage, XGetSubImage - transfer images

SYNOPSIS

```
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;

XImage *XGetImage(display, d, x, y, width, height, plane_mask, format)
        Display *display;
        Drawable d;
        int x, y;
        unsigned int width, height;
        long plane_mask;
        int format;

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;
```

ARGUMENTS

d	Specifies the drawable.
dest_image	Specify the destination image.
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 or 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.
display	Specifies the connection to the X server.
format	Specifies the format for the image. You can pass *XYBitmap, XYPixmap, or ZPixmap.*
gc	Specifies the GC.
image	Specifies the image you want combined with the rectangle.
plane_mask	Specifies the plane mask.
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.
width *height*	Specify the width and height of the subimage, which define the dimensions of the rectangle.

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.

DESCRIPTION

The *XPutImage* function combines an image in memory with a rectangle of the specified drawable. If *XYBitmap* format is used, the depth 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 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.

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 .I 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 noninferior 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.

XGetImage can generate *BadDrawable,* BadMatch, *and BadValue* errors.

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 noninferior 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.

XGetSubImage can generate *BadDrawable, BadGC, BadMatch,* and *BadValue* errors.

DIAGNOSTICS

BadDrawable A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch An *InputOnly* window is used as a Drawable.

BadMatch Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue Some numeric value falls outside 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 can generate this error.

NAME

XrmPutResource, XrmQPutResource, XrmPutStringResource, XrmQPutStringResource, XrmPutLineResource - store database resources

SYNOPSIS

```
void XrmPutResource(database, specifier, type, value)
     XrmDatabase *database;
     char *specifier;
     char *type;
     XrmValue *value;

void XrmQPutResource(database, bindings, quarks, type, value)
     XrmDatabase *database;
     XrmBindingList bindings;
     XrmQuarkList quarks;
     XrmRepresentation type;
     XrmValue *value;

void XrmPutStringResource(database, specifier, value)
     XrmDatabase *database;
     char *specifier;
     char *value;

void XrmQPutStringResource(database, bindings, quarks, value)
     XrmDatabase *database;
     XrmBindingList bindings;
     XrmQuarkList quarks;
     char *value;

void XrmPutLineResource(database, line)
     XrmDatabase *database;
     char *line;
```

ARGUMENTS

bindings	Specifies a list of bindings.
database	Specifies a pointer to the resource database.
line	Specifies the resource value pair as a single string. A single colon (:) separates the name from the value.
quarks	Specifies the complete or partial name or the class list of the resource.
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.

DESCRIPTION

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 database contains NULL, *XrmQPutResource* creates a new database and returns a pointer to it.

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 routine that takes both the resource and value as null-terminated strings, converts them to quarks, then calls *XrmQPutResource,* using a "String" representation type.

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.

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. Any white space before or after the name or colon in the line argument is ignored. The value is terminated by a new-line or a NULL character. To allow values to contain embedded new-line characters, a "\n" is recognized and replaced by a new-line character. For example, line might have the value "xterm*background:green\n". Null-terminated strings without a new line are also permitted.

SEE ALSO

XrmGetResource(3X11), XrmInitialize(3X11), XrmMergeDatabases(3X11), XrmUniqueQuark(3X11)

NAME

XQueryBestSize, XQueryBestTile, XQueryBestStipple - determine efficient sizes

SYNOPSIS

```
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;

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;

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;
```

ARGUMENTS

class	Specifies the class that you are interested in. You can pass *TileShape*, *CursorShape*, or *StippleShape*.
display	Specifies the connection to the X server.
width *height*	Specify the width and height.
which_screen	Specifies any drawable on the screen.
width_return *height_return*	Return the width and height of the object best supported by the display hardware.

DESCRIPTION

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.

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.

XQueryBestTile can generate *BadDrawable* and *BadMatch* errors.

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.

DIAGNOSTICS

BadMatch — An *InputOnly* window is used as a Drawable.

BadDrawable — A value for a Drawable argument does not name a defined Window or Pixmap.

BadMatch — The values do not exist for an *InputOnly* window.

BadValue — Some numeric value falls outside 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 can generate this error.

SEE ALSO

XCreateGC(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetState(3X11), XSetTile(3X11)

NAME

XQueryColor, XQueryColors, XLookupColor - obtain color values

SYNOPSIS

XQueryColor(display, colormap, def_in_out)
Display *display;
Colormap colormap;
XColor *def_in_out;

XQueryColors(display, colormap, defs_in_out, ncolors)
Display *display;
Colormap colormap;
XColor defs_in_out[];
int ncolors;

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;

ARGUMENTS

colormap	Specifies the colormap.
color_name	Specifies the color name string (for example, red) whose color definition structure you want returned.
def_in_out	Specifies and returns the RGB values for the pixel specified in the structure.
defs_in_out	Specifies and returns an array of color definition structures for the pixel specified in the structure.
display	Specifies the connection to the X server.
exact_def_return	Returns the exact RGB values.
ncolors	Specifies the number of *XColor* structures in the color definition array.
screen_def_return	Returns the closest RGB values provided by the hardware.

DESCRIPTION

The *XQueryColor* function returns the RGB values for each pixel in the *XColor* structures and sets the *DoRed, DoGreen,* and *DoBlue* flags. The *XQueryColors* function returns the RGB values for each pixel in the *XColor* structures and sets the *DoRed, DoGreen,* and *DoBlue* flags.

XQueryColor and *XQueryColors* can generate *BadColor* and *BadValue* errors.

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. You should use the ISO Latin-1 encoding; uppercase and lowercase do not matter. *XLookupColor* returns nonzero if the name existed in the color database or zero if it did not exist.

DIAGNOSTICS

BadColor	A value for a Colormap argument does not name a defined Colormap.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XAllocColor(3X11), XCreateColormap(3X11), XStoreColors(3X11)

NAME

XQueryPointer - get pointer coordinates

SYNOPSIS

```
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;
```

ARGUMENTS

child_return	Returns the child window that the pointer is located in, if any.
display	Specifies the connection to the X server.
mask_return	Returns the current state of the modifier keys and pointer buttons.
root_return	Returns the root window that the pointer is in.
root_x_return *root_y_return*	Return the pointer coordinates relative to the root window's origin.
w	Specifies the window.
win_x_return *win_y_return*	Return the pointer coordinates relative to the specified window.

DESCRIPTION

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.

XQueryPointer can generate a *BadWindow* error.

DIAGNOSTICS

BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XGetWindowAttributes(3X11), XQueryTree(3X11)

NAME

XQueryTree - query window tree information

SYNOPSIS

```
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;
```

ARGUMENTS

children_return	Returns a pointer to the list of children.
display	Specifies the connection to the X server.
nchildren_return	Returns the number of children.
parent_return	Returns the parent window.
root_return	Returns the root window.
w	Specifies the window whose list of children, root, parent, and number of children you want to obtain.

DESCRIPTION

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 nonzero if it succeeds. To free this list when it is no longer needed, use *XFree.*

NOTES

This really should return a screen *, not a root window ID.

SEE ALSO

XGetWindowAttributes(3X11), XQueryPointer(3X11)

NAME

XRaiseWindow, XLowerWindow, XCirculateSubwindows, XCirculateSubwindowsUp, XCirculateSubwindowsDown, XRestackWindows - change window stacking order

SYNOPSIS

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

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

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

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

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

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

ARGUMENTS

direction	Specifies the direction (up or down) that you want to circulate the window. You can pass *RaiseLowest* or *LowerHighest*.
display	Specifies the connection to the X server.
nwindows	Specifies the number of windows to be restacked.
w	Specifies the window.
windows	Specifies an array containing the windows to be restacked.

DESCRIPTION

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.

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.

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.

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.

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.

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 *BadWindow* error.

DIAGNOSTICS

BadValue — Some numeric value falls outside 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 can generate this error.

BadWindow — A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(3X11), XConfigureWindow(3X11), XCreateWindow(3X11), XDestroyWindow(3X11), XMapWindow(3X11), XUnmapWindow(3X11)

NAME

XReadBitmapFile, XWriteBitmapFile, XCreatePixmapFromBitmapData, XCreateBitmapFromData - manipulate bitmaps

SYNOPSIS

```
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;

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;

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;

Pixmap XCreateBitmapFromData(display, d, data, width, height)
      Display *display;
      Drawable d;
      char *data;
      unsigned int width, height;
```

ARGUMENTS

bitmap	Specifies the bitmap.
bitmap_return	Returns the bitmap that is created.
d	Specifies the drawable that indicates the screen.
data	Specifies the data in bitmap format.
data	Specifies the location of the bitmap data.
depth	Specifies the depth of the pixmap.
display	Specifies the connection to the X server.
fg *bg*	Specify the foreground and background pixel values to use.
filename	Specifies the file name to use. The format of the file name is operating-system dependent.
width *height*	Specify the width and height.
width_return *height_return*	Return the width and height values of the read in bitmap file.
x_hot *y_hot*	Specify where to place the hotspot coordinates (or -1,-1 if none are present) in the file.

x_hot_return
y_hot_return Return the hotspot coordinates.

DESCRIPTION

The *XReadBitmapFile* function reads in a file containing a bitmap. The file can be either in the standard X version 10 format (that is, the format used by X version 10 bitmap program) or in the X version 11 bitmap format. 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.

The *XWriteBitmapFile* function writes a bitmap out to a file. While *XReadBitmapFile* can read in either X version 10 format or X version 11 format, *XWriteBitmapFile* always writes out X version 11 format. 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.

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.

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.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.

BadDrawable A value for a Drawable argument does not name a defined Window or Pixmap.

BadMatch An *InputOnly* window is used as a Drawable.

NAME

XRecolorCursor, XFreeCursor, XQueryBestCursor - manipulate cursors

SYNOPSIS

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

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

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

ARGUMENTS

background_color	Specifies the RGB values for the background of the source.
cursor	Specifies the cursor.
d	Specifies the drawable, which indicates the screen.
display	Specifies the connection to the X server.
foreground_color	Specifies the RGB values for the foreground of the source.
width *height*	Specify the width and heightof 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.

DESCRIPTION

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.

XRecolorCursor can generate a *BadCursor* error.

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.

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.

DIAGNOSTICS

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.

SEE ALSO

XCreateFontCursor(3X11), XDefineCusor(3X11)

NAME

XReparentWindow - reparent windows

SYNOPSIS

XReparentWindow(display, w, parent, x, y)
Display *display;
Window w;
Window parent;
int x, y;

ARGUMENTS

display	Specifies the connection to the X server.
parent	Specifies the parent window.
w	Specifies the window.
x	
y	Specify the x and y coordinatesof the position in the new parent window.

DESCRIPTION

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 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.

DIAGNOSTICS

BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeSaveSet(3X11)

NAME

XrmGetResource, XrmQGetResource, XrmQGetSearchList, XrmQGetSearchResource - retrieve database resources and search lists

SYNOPSIS

```
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;

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;

typedef XrmHashTable *XrmSearchList;

Bool XrmQGetSearchList(database, names, classes, list_return, list_length)
      XrmDatabase database;
      XrmNameList names;
      XrmClassList classes;
      XrmSearchList list_return;
      int list_length;

Bool XrmQGetSearchResource(list, name, class, type_return, value_return)
      XrmSearchList list;
      XrmName name;
      XrmClass class;
      XrmRepresentation *type_return;
      XrmValue *value_return;
```

ARGUMENTS

class	Specifies the resource class.
classes	Specifies a list of resource classes.
database	Specifies the database that is to be used.
list	Specifies the search list returned by *XrmQGetSearchList.*
list_length	Specifies the number of entries (not the byte size) allocated for list_return.
list_return	Returns a search list for further use.
name	Specifies the resource name.
names	Specifies a list of resource names.
quark_class	Specifies the fully qualified class of the value being retrieved (as a quark).
quark_name	Specifies the fully qualified name of the value being retrieved (as a quark).
quark_type_return	Returns a pointer to the representation type of the destination (as a quark).
str_class	Specifies the fully qualified class of the value being retrieved (as a string).
str_name	Specifies the fully qualified name of the value being retrieved (as a string).
str_type_return	Returns a pointer to the representation type of the destination (as a string).
type_return	Returns data representation type.
value_return	Returns the value in the database.

DESCRIPTION

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.*

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.*

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.

SEE ALSO

XrmInitialize(3X11), XrmMergeDatabases(3X11), XrmPutResource(3X11), XrmUniqueQuark(3X11)

NAME

XrmInitialize, XrmParseCommand - initialize the Resource Manager and parse the command line

SYNOPSIS

```
void XrmInitialize();

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;
```

ARGUMENTS

argc_in_out	Specifies the number of arguments and returns the number of remaining arguments.
argv_in_out	Specifies a pointer to the command line arguments and returns the remaining arguments.
database	Specifies a pointer to the resource database.
name	Specifies the application name.
table	Specifies the table of command line arguments to be parsed.
table_count	Specifies the number of entries in the table.

DESCRIPTION

The *XrmInitialize* function initialize the resource manager.

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.

The specified table is used to parse the command line. Recognized entries in the table are removed from argv, and entries are made in 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 argc argument specifies the number of arguments in argv and is set 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 the specification. No separating (binding) character is inserted. 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.

SEE ALSO

XrmGetResource(3X11), XrmMergeDatabases(3X11), XrmPutResource(3X11), XrmUniqueQuark(3X11)

NAME

XSaveContext, XFindContext, XDeleteContext, XUniqueContext - associative look-up routines

SYNOPSIS

```
int XSaveContext(display, w, context, data)
      Display *display;
      Window w;
      XContext context;
      caddr_t data;

int XFindContext(display, w, context, data_return)
      Display *display;
      Window w;
      XContext context;
      caddr_t *data_return;

int XDeleteContext(display, w, context)
      Display *display;
      Window w;
      XContext context;

XContext XUniqueContext()
```

ARGUMENTS

context	Specifies the context type to which the data belongs.
data	Specifies the data to be associated with the window and type.
data_return	Returns a pointer to the data.
display	Specifies the connection to the X server.
w	Specifies the window with which the data is associated.

DESCRIPTION

If an entry with the specified window 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).

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).

The *XDeleteContext* function deletes the entry for the given window 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.

The *XUniqueContext* function creates a unique context type that may be used in subsequent calls to *XSaveContext*.

NAME

XSetFont - GC convience routines

SYNOPSIS

XSetFont (display, gc, font)
Display *display;
GC gc;
Font font;

ARGUMENTS

display	Specifies the connection to the X server.
font	Specifies the font.
gc	Specifies the GC.

DESCRIPTION

The *XSetFont* function sets the current font in the specified GC.

XSetFont can generate *BadAlloc, BadFont,* and *BadGC* errors.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadFont	A value for a Font or GContext argument does not name a defined Font.
BadGC	A value for a GContext argument does not name a defined GContext.

SEE ALSO

XCreateGC(3X11), XQueryBestSize(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetLineAttributes(3X11), XSetState(3X11), XSetTile(3X11)

NAME

XSetFontPath, XGetFontPath, XFreeFontPath - set, get, or free the font search path

SYNOPSIS

```
XSetFontPath(display, directories, ndirs)
      Display *display;
      char **directories;
      int ndirs;

char **XGetFontPath(display, npaths_return)
      Display *display;
      int *npaths_return;

XFreeFontPath(list)
      char **list;
```

ARGUMENTS

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.
display	Specifies the connection to the X server.
list	Specifies the array of strings you want to free.
ndirs	Specifies the number of directories in the path.
npaths_return	Returns the number of strings in the font path array.

DESCRIPTION

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 interpretation of the strings is operating system dependent, but they are intended to specify directories to be searched in the order listed. Also, the contents of these strings are operating system dependent and are not intended to be used by client applications. Usually, the X server is free to cache font information internally rather than having to read fonts from files. In addition, 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 operating system dependent.

XSetFontPath can generate a *BadValue* error.

The *XGetFontPath* function allocates and returns an array of strings containing the search path. When it is no longer needed, the data in the font path should be freed by using *XFreeFontPath.*

The *XFreeFontPath* function frees the data allocated by *XGetFontPath.*

DIAGNOSTICS

BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XListFont(3X11), XLoadFonts(3X11)

NAME

XSetIconName, XGetIconName - set or get icon names

SYNOPSIS

```
XSetIconName(display, w, icon_name)
      Display *display;
      Window w;
      char *icon_name;

Status XGetIconName(display, w, icon_name_return)
      Display *display;
      Window w;
      char **icon_name_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
icon_name	Specifies the icon name, which should be a null-terminated string.
icon_name_return	Returns a pointer to the window's icon name, which is a null-terminated string.
w	Specifies the window.

DESCRIPTION

The *XSetIconName* function sets the name to be displayed in a window's icon.

XSetIconName can generate *BadAlloc* and *BadWindow* errors.

The *XGetIconName* function returns the name to be displayed in the specified window's icon. If it succeeds, it returns nonzero; 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. When finished with it, a client must free the icon name string using *XFree.*

XGetIconName can generate a *BadWindow* error.

PROPERTY

WM_ICON_NAME

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetCommand(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetTransientForHint(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetIconSizes, XGetIconSizes - set or get icon size hints

SYNOPSIS

```
XSetIconSizes(display, w, size_list, count)
      Display *display;
      Window w;
      XIconSize *size_list;
      int count;

Status XGetIconSizes(display, w, size_list_return, count_return)
      Display *display;
      Window w;
      XIconSize **size_list_return;
      int *count_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
count	Specifies the number of items in the size list.
count_return	Returns the number of items in the size list.
size_list	Specifies a pointer to the size list.
size_list_return	Returns a pointer to the size list.
w	Specifies the window.

DESCRIPTION

The *XSetIconSizes* function is used only by window managers to set the supported icon sizes.

XSetIconSizes can generate *BadAlloc* and *BadWindow* errors.

The *XGetIconSizes* function returns zero if a window manager has not set icon sizes or nonzero otherwise. *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.

PROPERTY

WM_ICON_SIZE

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetCommand(3X11), XSetIconName(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetTransientForHint(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetInputFocus, XGetInputFocus - control input focus

SYNOPSIS

```
XSetInputFocus(display, focus, revert_to, time)
      Display *display;
      Window focus;
      int revert_to;
      Time time;

XGetInputFocus(display, focus_return, revert_to_return)
      Display *display;
      Window *focus_return;
      int *revert_to_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
focus	Specifies the window, *PointerRoot,* or *None.*
focus_return	Returns the focus 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.*
revert_to_return	Returns the current focus state (RevertToParent, *RevertToPointerRoot,* or *RevertToNone*).
time	Specifies the time. You can pass either a timestamp or *CurrentTime.*

DESCRIPTION

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.

The *XGetInputFocus* function returns the focus window and the current focus state.

DIAGNOSTICS

BadValue Some numeric value falls outside 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 can generate this error.

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XWarpPointer(3X11)

NAME

XSetLineAttribute, XSetDashes - GC convience routines

SYNOPSIS

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;

XSetDashes(display, gc, dash_offset, dash_list, n)
Display *display;
GC gc;
int dash_offset;
char dash_list[];
int n;

ARGUMENTS

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.*
dash_list	Specifies the dash-list for the dashed line-style you want to set for the specified GC.
dash_offset	Specifies the phase of the pattern for the dashed line-style you want to set for the specified GC.
display	Specifies the connection to the X server.
gc	Specifies the GC.
join_style	Specifies the line join-style you want to set for the specified GC. You can pass *JoinMiter, JoinRound,* or *JoinBevel.*
line_style	Specifies the line-style you want to set for the specified GC. You can pass *LineSolid, LineOnOffDash,* or *LineDoubleDash.*
line_width	Specifies the line-width you want to set for the specified GC.
n	Specifies the number of elements in dash_list.

DESCRIPTION

The *XSetLineAttributes* function sets the line drawing components in the specified GC.

XSetLineAttributes can generate *BadAlloc, BadGC,* and *BadValue* errors.

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 nonzero, 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 each time a cap-style is applied at a line endpoint.

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 315 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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadGC	A value for a GContext argument does not name a defined GContext.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XCreateGC(3X11), XQueryBestSize(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetState(3X11), XSetTile(3X11)

NAME

XSetNormalHints, XGetNormalHints - set or get normal state hints

SYNOPSIS

```
XSetNormalHints(display, w, hints)
      Display *display;
      Window w;
      XSizeHints *hints;

Status XGetNormalHints(display, w, hints_return)
      Display *display;
      Window w;
      XSizeHints *hints_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
hints	Specifies a pointer to the size hints for the window in its normal state.
hints_return	Returns the size hints for the window in its normal state.
w	Specifies the window.

DESCRIPTION

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.

The *XGetNormalHints* function returns the size hints for a window in its normal state. It returns a nonzero status if it succeeds or zero if the application specified no normal size hints for this window.

XGetNormalHints can generate a *BadWindow* error.

PROPERTY

WM_NORMAL_HINTS

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetPointerMapping, XGetPointerMapping - manipulate pointer settings

SYNOPSIS

```
int XSetPointerMapping(display, map, nmap)
      Display *display;
      unsigned char map[];
      int nmap;

int XGetPointerMapping(display, map_return, nmap)
      Display *display;
      unsigned char map_return[];
      int nmap;
```

ARGUMENTS

display	Specifies the connection to the X server.
map	Specifies the mapping list.
map_return	Returns the mapping list.
nmap	Specifies the number of items in the mapping list.

DESCRIPTION

The *XSetPointerMapping* function sets the mapping of the pointer. If it succeeds, the X server generates a *MappingNotify* event, and *XSetPointerMapping* returns *MappingSuccess*. Elements of the list are indexed starting from one. The length of the list must be the same as *XGetPointerMapping* would return, or a *BadValue* error results. The index is a core button number, and the element of the list defines the effective number. 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 nonzero 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.

The *XGetPointerMapping* function returns the current mapping of the pointer. Elements of the list are indexed starting from one. *XGetPointerMapping* returns the number of physical buttons actually on the pointer. The nominal mapping for a pointer is the identity mapping: map[i]=i. 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.

DIAGNOSTICS

BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XChangeKeyboardControl(3X11), XChangeKeyboardMapping(3X11)

NAME

XSetScreenSaver, XForceScreenSaver, XActivateScreenSaver, XResetScreenSaver, XGetScreenSaver - manipulate the screen saver

SYNOPSIS

```
XSetScreenSaver(display, timeout, interval, prefer_blanking, allow_exposures)
      Display *display;
      int timeout, interval;
      int prefer_blanking;
      int allow_exposures;

XForceScreenSaver(display, mode)
      Display *display;
      int mode;

XActivateScreenSaver(display)
      Display *display;

XResetScreenSaver(display)
      Display *display;

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;
```

ARGUMENTS

allow_exposures Specifies the screen save control values. You can pass *DontAllowExposures, AllowExposures,* or *DefaultExposures.*

allow_exposures_return
Returns the current screen save control value (**DontAllowExposures,** *AllowExposures,* or *DefaultExposures*).

display Specifies the connection to the X server.

interval Specifies the interval between screen saver alterations.

interval_return Returns the interval between screen saver invocations.

mode Specifies the mode that is to be applied. You can pass *ScreenSaverActive* or *ScreenSaverReset.*

prefer_blanking Specifies how to enable screen blanking. You can pass *DontPreferBlanking, PreferBlanking,* or *DefaultBlanking.*

prefer_blanking_return
Returns the current screen blanking preference (**DontPreferBlanking,** *PreferBlanking,* or *DefaultBlanking*).

timeout Specifies the timeout, in seconds, until the screen saver turns on.

timeout_return Returns the timeout, in minutes, until the screen saver turns on.

DESCRIPTION

Timeout and interval are specified in seconds. A timeout of 0 disables the screen saver, 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.

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.

The *XActivateScreenSaver* function activates the screen saver.

The *XResetScreenSaver* function resets the screen saver.

The *XGetScreenSaver* function gets the current screen saver values.

DIAGNOSTICS

BadValue Some numeric value falls outside 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 can generate this error.

NAME

XSelectInput - select input events

SYNOPSIS

XSelectInput (display, w, event_mask)
Display *display;
Window w;
long event_mask;

ARGUMENTS

display	Specifies the connection to the X server.
event_mask	Specifies the event mask.
w	Specifies the window whose events you are interested in.

DESCRIPTION

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 client. 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.

DIAGNOSTICS

BadWindow	A value for a Window argument does not name a defined Window.

NAME

XSetArcMode, XSetSubwindowMode, XSetGraphicsExposure - GC convience routines

SYNOPSIS

```
XSetArcMode(display, gc, arc_mode)
      Display *display;
      GC gc;
      int arc_mode;

XSetSubwindowMode(display, gc, subwindow_mode)
      Display *display;
      GC gc;
      int subwindow_mode;

XSetGraphicsExposures(display, gc, graphics_exposures)
      Display *display;
      GC gc;
      Bool graphics_exposures;
```

ARGUMENTS

arc_mode	Specifies the arc mode. You can pass *ArcChord* or *ArcPieSlice.*
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.
subwindow_mode	Specifies the subwindow mode. You can pass *ClipByChildren* or *IncludeInferiors.*

DESCRIPTION

The *XSetArcMode* function sets the arc mode in the specified GC.

XSetArcMode can generate *BadAlloc, BadGC,* and *BadValue* errors.

The *XSetSubwindowMode* function sets the subwindow mode in the specified GC.

XSetSubwindowMode can generate *BadAlloc, BadGC,* and *BadValue* errors.

The *XSetGraphicsExposures* function sets the graphics-exposures flag in the specified GC.

XSetGraphicsExposures can generate *BadAlloc, BadGC,* and *BadValue* errors.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadGC	A value for a GContext argument does not name a defined GContext.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XCreateGC(3X11), XQueryBestSize(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetState(3X11), XSetTile(3X11)

NAME

XSetClassHint, XGetClassHint - set or get class hint

SYNOPSIS

```
XSetClassHint(display, w, class_hints)
      Display *display;
      Window w;
      XClassHint *class_hints;

Status XGetClassHint(display, w, class_hints_return)
      Display *display;
      Window w;
      XClassHint *class_hints_return;
```

ARGUMENTS

class_hints	Specifies a pointer to a *XClassHint* structure that is to be used.
class_hints_return	Returns the *XClassHint* structure.
display	Specifies the connection to the X server.
w	Specifies the window.

DESCRIPTION

The *XSetClassHint* function sets the class hint for the specified window.

XSetClassHint can generate *BadAlloc* and *BadWindow* errors.

The *XGetClassHint* function returns the class of the specified window. To free res_name and res_class when finished with the strings, use *XFree.*

XGetClassHint can generate a *BadWindow* error.

PROPERTY

WM_CLASS

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetTransientForHint(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetClipOrigin, XSetClipMask, XSetClipRectangles - GC convience routines

SYNOPSIS

```
XSetClipOrigin(display, gc, clip_x_origin, clip_y_origin)
      Display *display;
      GC gc;
      int clip_x_origin, clip_y_origin;

XSetClipMask(display, gc, pixmap)
      Display *display;
      GC gc;
      Pixmap pixmap;

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;
```

ARGUMENTS

display	Specifies the connection to the X server.
clip_x_origin *clip_y_origin*	Specify the x and y coordinates of the clip-mask origin.
gc	Specifies the GC.
n	Specifies the number of rectangles.
ordering	Specifies the ordering relations on the rectangles. You can pass *Unsorted*, *YSorted*, *YXSorted*, or *YXBanded*.
pixmap	Specifies the pixmap or *None*
rectangles	Specifies an array of rectangles that define the clip-mask.

DESCRIPTION

The *XSetClipOrigin* function sets the clip origin in the specified GC. 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.

The *XSetClipMask* function sets the clip-mask in the specified GC to the specified pixmap. 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 *BadValue* errors.

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 nondecreasing in their Y origin. *YXSorted* additionally constrains *YSorted* order in that all rectangles with an equal Y origin are nondecreasing 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.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XCreateGC(3X11), XQueryBestSize(3X11), XSetArcMode(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetState(3X11), XSetTile(3X11)

NAME

XSetCloseDownMode, XKillClient - control clients

SYNOPSIS

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

XKillClient(display, resource)
      Display *display;
      XID resource;
```

ARGUMENTS

close_mode	Specifies the client close-down mode. You can pass *DestroyAll*, *RetainPermanent*, or *RetainTemporary*.
display	Specifies the connection to the X server.
resource	Specifies any resource associated with the client that you want to destroy or *AllTemporary*.

DESCRIPTION

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 2.6.

XSetCloseDownMode can generate a *BadValue* error.

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 2.6). 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.

DIAGNOSTICS

BadValue	Some numeric value falls outside 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 can generate this error.

NAME

XSetCommand - set command atom

SYNOPSIS

```
XSetCommand(display, w, argv, argc)
      Display *display;
      Window w;
      char **argv;
      int argc;
```

ARGUMENTS

argc	Specifies the number of arguments.
argv	Specifies the application's argument list.
display	Specifies the connection to the X server.
w	Specifies the window.

DESCRIPTION

The *XSetCommand* function sets the command and arguments used to invoke the application. (Typically, argv is the argv array of your main program.)

XSetCommand can generate *BadAlloc* and *BadWindow* errors.

PROPERTY

WM_COMMAND

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetTransientForHint(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetErrorHandler, XGetErrorText, XDisplayName, XSetIOErrorHandler, XGetErrorDatabaseText - default error handlers

SYNOPSIS

```
XSetErrorHandler(handler)
      int (*handler)(Display *, XErrorEvent *)

XGetErrorText(display, code, buffer_return, length)
      Display *display;
      int code;
      char *buffer_return;
      int length;

char *XDisplayName(string)
      char *string;

XSetIOErrorHandler(handler)
      int (*handler)(Display *);

XGetErrorDatabaseText(display, name, message, default_string, buffer_return, length)
      Display *display;
      char *name, *message;
      char *default_string;
      char *buffer_return;
      int length;
```

ARGUMENTS

buffer_return	Returns the error description.
code	Specifies the error code for which you want to obtain a description.
default_string	Specifies the default error message if none is found in the database.
display	Specifies the connection to the X server.
handler	Specifies the program's supplied error handler.
length	Specifies the size of the buffer.
message	Specifies the type of the error message.
name	Specifies the name of the application.
string	Specifies the character string.

DESCRIPTION

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 *XGetErrorText* function copies a null-terminated string describing the specified error code into the specified buffer. 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.

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.

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.

The *XGetErrorDatabaseText* function returns a message (or the default message) from the error message database. Xlib uses this function internally to look up its error messages. On a UNIX-based system, the error message database is */usr/lib/X11/XErrorDB*.

The name argument should generally be the name of your application. The message argument should indicate which type of error message you want. Xlib uses three predefined message types to report errors (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	The major 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.

SEE ALSO

XSynchronize(3X11)

NAME

XSendEvent, XDisplayMotionBufferSize, XGetMotionEvents - send events

SYNOPSIS

```
Status XSendEvent(display, w, propagate, event_mask, event_send)
      Display *display;
      Window w;
      Bool propagate;
      long event_mask;
      XEvent *event_send;

unsigned long XDisplayMotionBufferSize(display)
      Display *display;

XTimeCoord *XGetMotionEvents(display, w, start, stop, nevents_return)
      Display *display;
      Window w;
      Time start, stop;
      int *nevents_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
event_mask	Specifies the event mask.
event_send	Specifies a pointer to the event that is to be sent.
nevents_return	Returns the number of events from the motion history buffer.
propagate	Specifies a Boolean value.
start *stop*	Specify the time interval in which the events are returned from the motion history buffer. You can pass a timestamp or *CurrentTime.*
w	Specifies the destination window.

DESCRIPTION

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 8.3. 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 nonzero otherwise. *XSendEvent* can generate *BadValue* and *BadWindow* errors.

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.

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 start time is later than the stop time or if the start time is in the future, no events are returned. If the stop time is in the future, it is equivalent to specifying *CurrentTime*. *XGetMotionEvents* can generate a *BadWindow* error.

DIAGNOSTICS

BadValue Some numeric value falls outside 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 can generate this error.

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XIfEvent(3X11), XNextEvent(3X11), XPutBackEvent(3X11)

NAME

XSetFillStyle, XSetFillRule - GC convience routines

SYNOPSIS

XSetFillStyle(display, gc, fill_style)
Display *display;
GC gc;
int fill_style;

XSetFillRule(display, gc, fill_rule)
Display *display;
GC gc;
int fill_rule;

ARGUMENTS

display Specifies the connection to the X server.

fill_rule Specifies the fill-rule you want to set for the specified GC. You can pass *EvenOddRule* or *WindingRule.*

fill_style Specifies the fill-style you want to set for the specified GC. You can pass *FillSolid, FillTiled, FillStippled,* or *FillOpaqueStippled.*

gc Specifies the GC.

DESCRIPTION

The *XSetFillStyle* function sets the fill-style in the specified GC.

XSetFillStyle can generate *BadAlloc, BadGC,* and *BadValue* errors.

The *XSetFillRule* function sets the fill-rule in the specified GC.

XSetFillRule can generate *BadAlloc, BadGC,* and *BadValue* errors.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.

BadGC A value for a GContext argument does not name a defined GContext.

BadValue Some numeric value falls outside 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 can generate this error.

SEE ALSO

XCreateGC(3X11), XQueryBestSize(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetState(3X11), XSetTile(3X11)

NAME

XSetSelectionOwner, XGetSelectionOwner, XConvertSelection - manipulate window selection

SYNOPSIS

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

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

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

ARGUMENTS

display	Specifies the connection to the X server.
owner	Specifies the owner of the specified selection atom. You can pass a window or *None.*
property	Specifies the property name. You also can pass *None.*
requestor	Specifies the requestor.
selection	Specifies the selection atom.
target	Specifies the target atom.
time	Specifies the time. You can pass either a timestamp or *CurrentTime.*

DESCRIPTION

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.

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.

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.*

In either event, the arguments are passed on unchanged. There are two predefined selection atoms: PRIMARY and SECONDARY.

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

DIAGNOSTICS

BadAtom	A value for an Atom argument does not name a defined Atom.
BadWindow	A value for a Window argument does not name a defined Window.

NAME

XSetSizeHints, XGetSizeHints - set or get window size hints

SYNOPSIS

XSetSizeHints(display, w, hints, property)
Display *display;
Window w;
XSizeHints *hints;
Atom property;

Status XGetSizeHints(display, w, hints_return, property)
Display *display;
Window w;
XSizeHints *hints_return;
Atom property;

ARGUMENTS

display	Specifies the connection to the X server.
hints	Specifies a pointer to the size hints.
hints_return	Returns the size hints.
property	Specifies the property name.
w	Specifies the window.

DESCRIPTION

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.

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 nonzero status if a size hint was defined or zero otherwise.

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

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadAtom	A value for an Atom argument does not name a defined Atom.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetStandardProperties(3X11), XSetTransientForHint(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetStandardColormap, XGetStandardColormap - set or get standard colormaps

SYNOPSIS

```
XSetStandardColormap(display, w, colormap, property)
      Display *display;
      Window w;
      XStandardColormap *colormap;
      Atom property;    /* RGB_BEST_MAP, etc. */

Status XGetStandardColormap(display, w, colormap_return, property)
      Display *display;
      Window w;
      XStandardColormap *colormap_return;
      Atom property;    /* RGB_BEST_MAP, etc. */
```

ARGUMENTS

colormap	Specifies the colormap.
colormap_return	Returns the colormap associated with the specified atom.
display	Specifies the connection to the X server.
property	Specifies the property name.
w	Specifies the window.

DESCRIPTION

The *XSetStandardColormap* function usually is only used by window 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 (not required for RGB_DEFAULT_MAP)
   - Determine the color capabilities of the display.
   - 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.

XSetStandardColormap can generate *BadAlloc, BadAtom,* and *BadWindow* errors.

The *XGetStandardColormap* function returns the colormap definition associated with the atom supplied as the property argument. 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);
```

Once you have fetched a standard colormap, you can use it to convert RGB values into pixel values. For example, given an *XStandardColormap* structure and floating-point RGB coefficients in the range 0.0 to 1.0, you can compose pixel values with the following C expression:

```
pixel = base_pixel
        + ((unsigned long) (0.5 + r * red_max)) * red_mult
        + ((unsigned long) (0.5 + g * green_max)) * green_mult
        + ((unsigned long) (0.5 + b * blue_max)) * blue_mult;
```

The use of addition rather than logical OR for composing pixel values permits allocations where the RGB value is not aligned to bit boundaries.

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

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadAtom	A value for an Atom argument does not name a defined Atom.
BadWindow	A value for a Window argument does not name a defined Window.

NAME

XSetStandardProperties - set standard window manager properties

SYNOPSIS

```
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;
```

ARGUMENTS

argc	Specifies the number of arguments.
argv	Specifies the application's argument list.
display	Specifies the connection to the X server.
hints	Specifies a pointer to the size hints for the window in its normal state.
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*
w	Specifies the window.
window_name	Specifies the window name, which should be a null-terminated string.

DESCRIPTION

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.)

XSetStandardProperties can generate *BadAlloc* and *BadWindow* errors.

PROPERTIES

WM_NAME, WM_ICON_NAME, WM_HINTS, WM_COMMAND, and WM_NORMALHINTS

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetTransientForHint(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetState, XSetFunction, XSetPlanemask, XSetForeground, XSetBackground - GC convience routines

SYNOPSIS

```
XSetState(display, gc, foreground, background, function, plane_mask)
      Display *display;
      GC gc;
      unsigned long foreground, background;
      int function;
      unsigned long plane_mask;

XSetFunction(display, gc, function)
      Display *display;
      GC gc;
      int function;

XSetPlaneMask(display, gc, plane_mask)
      Display *display;
      GC gc;
      unsigned long plane_mask;

XSetForeground(display, gc, foreground)
      Display *display;
      GC gc;
      unsigned long foreground;

XSetBackground(display, gc, background)
      Display *display;
      GC gc;
      unsigned long background;
```

ARGUMENTS

background	Specifies the background you want to set for the specified GC.
display	Specifies the connection to the X server.
foreground	Specifies the foreground you want to set for the specified GC.
function	Specifies the function you want to set for the specified GC.
gc	Specifies the GC.
plane_mask	Specifies the plane mask.

DESCRIPTION

The *XSetState* function sets the foreground, background, plane mask, and function components for the specified GC.

XSetState can generate *BadAlloc, BadGC,* and *BadValue* errors.

XSetFunction sets a specified value in the specified GC.

XSetFunction can generate *BadAlloc, BadGC,* and *BadValue* errors.

The *XSetPlaneMask* function sets the plane mask in the specified GC.

XSetPlaneMask can generate *BadAlloc* and *BadGC* errors.

The *XSetForeground* function sets the foreground in the specified GC.

XSetForeground can generate *BadAlloc* and *BadGC* errors.

The *XSetBackground* function sets the background in the specified GC.

XSetBackground can generate *BadAlloc* and *BadGC* errors.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.

BadGC	A value for a GContext argument does not name a defined GContext.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XCreateGC(3X11), XQueryBestSize(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetTile(3X11)

NAME

XSetTile, XSetStipple, XSetTSOrigin - GC convience routines

SYNOPSIS

```
XSetTile(display, gc, tile)
      Display *display;
      GC gc;
      Pixmap tile;

XSetStipple(display, gc, stipple)
      Display *display;
      GC gc;
      Pixmap stipple;

XSetTSOrigin(display, gc, ts_x_origin, ts_y_origin)
      Display *display;
      GC gc;
      int ts_x_origin, ts_y_origin;
```

ARGUMENTS

display	Specifies the connection to the X server.
gc	Specifies the GC.
stipple	Specifies the stipple you want to set for the specified GC.
tile	Specifies the fill tile you want to set for the specified GC.
ts_x_origin *ts_y_origin*	Specify the x and y coordinates of the tile and stipple origin.

DESCRIPTION

The *XSetTile* function sets the fill tile in 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.

The *XSetStipple* function sets the stipple in the specified GC. The stipple and GC must have the same depth, or a *BadMatch* error results.

XSetStipple can generate *BadAlloc, BadGC, BadMatch,* and *BadPixmap* errors.

The *XSetTSOrigin* function sets the tile/stipple origin in the specified GC. 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* errors.

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadGC	A value for a GContext argument does not name a defined GContext.
BadMatch	Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
BadPixmap	A value for a Pixmap argument does not name a defined Pixmap.

SEE ALSO

XCreateGC(3X11), XQueryBestSize(3X11), XSetArcMode(3X11), XSetClipOrigin(3X11), XSetFillStyle(3X11), XSetFont(3X11), XSetLineAttributes(3X11), XSetState(3X11)

NAME

XSetTransientForHint, XGetTransientForHint - set or get transient for hint

SYNOPSIS

XSetTransientForHint(display, w, prop_window)
Display *display;
Window w;
Window prop_window;

Status XGetTransientForHint(display, w, prop_window_return)
Display *display;
Window w;
Window *prop_window_return;

ARGUMENTS

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.
prop_window_return	Returns the WM_TRANSIENT_FOR property of the specified window.

DESCRIPTION

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.

The *XGetTransientForHint* function returns the WM_TRANSIENT_FOR property for the specified window.

XGetTransientForHint can generate a *BadWindow* error.

PROPERTY

WM_TRANSIENT_FOR

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetWMHints(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetWMHints, XGetWMHints - set or get window manager hints

SYNOPSIS

```
XSetWMHints(display, w, wmhints)
      Display *display;
      Window w;
      XWMHints *wmhints;

XWMHints *XGetWMHints(display, w)
      Display *display;
      Window w;
```

ARGUMENTS

display	Specifies the connection to the X server.
w	Specifies the window.
wmhints	Specifies a pointer to the window manager hints.

DESCRIPTION

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.

The *XGetWMHints* function reads the window manager hints and returns NULL if no WM_HINTS property was set on the window or 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.

PROPERTY

WM_HINTS

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetTransientForHint(3X11), XSetZoomHints(3X11), XStoreName(3X11)

NAME

XSetZoomHints, XGetZoomHints - set or get zoom state hints

SYNOPSIS

```
XSetZoomHints(display, w, zhints)
      Display *display;
      Window w;
      XSizeHints *zhints;

Status XGetZoomHints(display, w, zhints_return)
      Display *display;
      Window w;
      XSizeHints *zhints_return;
```

ARGUMENTS

display	Specifies the connection to the X server.
w	Specifies the window.
zhints	Specifies a pointer to the zoom hints.
zhints_return	Returns the zoom hints.

DESCRIPTION

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.

The *XGetZoomHints* function returns the size hints for a window in its zoomed state. It returns a nonzero status if it succeeds or zero if the application specified no zoom size hints for this window.

XGetZoomHints can generate a *BadWindow* error.

PROPERTY

WM_ZOOM_HINTS

DIAGNOSTICS

BadAlloc	The server failed to allocate the requested resource or server memory.
BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XSetClassHint(3X11), XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetTransientForHint(3X11), XSetWMHints(3X11), XStoreName(3X11)
Xlib - C Language X Interface

NAME

XStoreBytes, XStoreBuffer, XFetchBytes, XFetchBuffer, XRotateBuffers - manipulate cut and paste buffers

SYNOPSIS

```
XStoreBytes(display, bytes, nbytes)
      Display *display;
      char *bytes;
      int nbytes;

XStoreBuffer(display, bytes, nbytes, buffer)
      Display *display;
      char *bytes;
      int nbytes;
      int buffer;

char *XFetchBytes(display, nbytes_return)
      Display *display;
      int *nbytes_return;

char *XFetchBuffer(display, nbytes_return, buffer)
      Display *display;
      int *nbytes_return;
      int buffer;

XRotateBuffers(display, rotate)
      Display *display;
      int rotate;
```

ARGUMENTS

buffer	Specifies the buffer in which you want to store the bytes or from which you want the stored data returned.
bytes	Specifies the bytes, which are not necessarily ASCII or null-terminated.
display	Specifies the connection to the X server.
nbytes	Specifies the number of bytes to be stored.
nbytes_return	Returns the number of bytes in the buffer.
rotate	Specifies how much to rotate the cut buffers.

DESCRIPTION

Note that the cut buffer's contents need not be text, so zero bytes are not special. The cut buffer's contents can be retrieved later by any client calling *XFetchBytes.*

XStoreBytes can generate a *BadAlloc* error.

If the property for the buffer has never been created, a *BadAtom* error results.

XStoreBuffer can generate *BadAlloc* and *BadAtom* errors.

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.* Note that the cut buffer does not necessarily contain text, so it may contain embedded zero bytes and may not terminate with a null byte.

The *XFetchBuffer* function returns zero to the nbytes_return argument if there is no data in the buffer.

XFetchBuffer can generate a *BadValue* error.

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.

XRotateBuffers can generate a *BadMatch* error.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.

BadAtom A value for an Atom argument does not name a defined Atom.

BadMatch Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue Some numeric value falls outside 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 can generate this error.

NAME

XStoreColors, XStoreColor, XStoreNamedColor - set colors

SYNOPSIS

```
XStoreColors(display, colormap, color, ncolors)
      Display *display;
      Colormap colormap;
      XColor color[];
      int ncolors;

XStoreColor(display, colormap, color)
      Display *display;
      Colormap colormap;
      XColor *color;

XStoreNamedColor(display, colormap, color, pixel, flags)
      Display *display;
      Colormap colormap;
      char *color;
      unsigned long pixel;
      int flags;
```

ARGUMENTS

color	Specifies the pixel and RGB values or the color name string (for example, red).
color	Specifies an array of color definition structures to be stored.
colormap	Specifies the colormap.
display	Specifies the connection to the X server.
flags	Specifies which red, green, and blue components are set.
ncolors	Specifies the number of *XColor* structures in the color definition array.
pixel	Specifies the entry in the colormap.

DESCRIPTION

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,* 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.

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,* 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.

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 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. You should use the ISO Latin-1 encoding; uppercase and lowercase do not matter.

XStoreNamedColor can generate *BadAccess, BadColor, BadName,* and *BadValue* errors.

DIAGNOSTICS

BadAccess	A client attempted to free a color map entry that it did not already allocate.
BadAccess	A client attempted to store into a read-only color map entry.
BadColor	A value for a Colormap argument does not name a defined Colormap.
BadName	A font or color of the specified name does not exist.
BadValue	Some numeric value falls outside 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 can generate this error.

SEE ALSO

XAllocColor(3X11), XCreateColormap(3X11), XQueryColor(3X11)

NAME

XStoreName, XFetchName - set or get window names

SYNOPSIS

```
XStoreName(display, w, window_name)
      Display *display;
      Window w;
      char *window_name;

Status XFetchName(display, w, window_name_return)
      Display *display;
      Window w;
      char **window_name_return;
```

ARGUMENTS

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.

window_name_return
Returns a pointer to the window name, which is a null-terminated string.

DESCRIPTION

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.

XStoreName can generate *BadAlloc* and *BadWindow* errors.

The *XFetchName* function returns the name of the specified window. If it succeeds, it returns nonzero; otherwise, if no name has been set for the window, it returns zero. If the WM_NAME property has not been set for this window, *XFetchName* sets window_name_return to NULL. When finished with it, a client must free the window name string using *XFree*.

XFetchName can generate a *BadWindow* error.

PROPERTY

WM_NAME

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XSetCommand(3X11), XSetIconName(3X11), XSetIconSizeHints(3X11), XSetNormalHints(3X11), XSetSizeHints(3X11), XSetStandardProperties(3X11), XSetWMHints(3X11), XSetZoomHints(3X11)

NAME

XStringToKeysym, XKeysymToString, XKeycodeToKeysym, XKeysymToKeycode - convert keysyms

SYNOPSIS

KeySym XStringToKeysym(string)
char *string;

char *XKeysymToString(keysym)
KeySym keysym;

KeySym XKeycodeToKeysym(display, keycode, index)
Display *display;
KeyCode keycode;
int index;

KeyCode XKeysymToKeycode(display, keysym)
Display *display;
KeySym keysym;

ARGUMENTS

display	Specifies the connection to the X server.
index	Specifies the element of KeyCode vector.
keycode	Specifies the KeyCode.
keysym	Specifies the KeySym that is to be searched for or converted.
string	Specifies the name of the KeySym that is to be converted.

DESCRIPTION

Valid KeySym names are listed in <**X11/keysymdef.h**> by removing the XK_ prefix from each name. If the specified string does not match a valid KeySym, *XStringToKeysym* returns *NoSymbol.*

The returned string is in a static area and must not be modified. If the specified KeySym is not defined, *XKeysymToString* returns a NULL.

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.*

If the specified KeySym is not defined for any KeyCode, *XKeysymToKeycode* returns zero.

SEE ALSO

XLookupKeysym(3X11)

NAME

XSynchronize, XSetAfterFunction - enable or disable synchronization

SYNOPSIS

```
int (*XSynchronize(display, onoff))()
      Display *display;
      Bool onoff;

int (*XSetAfterFunction(display, procedure))()
      Display *display;
      int (*procedure)();
```

ARGUMENTS

display	Specifies the connection to the X server.
procedure	Specifies the function to be called after an Xlib function that generates a protocol request completes its work.
onoff	Specifies a Boolean value that indicates whether to enable or disable synchronization.

DESCRIPTION

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.

The specified procedure is called with only a display pointer. *XSetAfterFunction* returns the previous after function.

SEE ALSO

XSetErrorHandler(3X11)

NAME

XTextExtents, XTextExtents16, XQueryTextExtents, XQueryTextExtents16 - compute or query text extents

SYNOPSIS

```
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;

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;

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;
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;
```

ARGUMENTS

direction_return	Returns the value of the direction hint (FontLeftToRight or *FontRightToLeft*).
display	Specifies the connection to the X server.
font_ID	Specifies either the font ID or the *GContext* ID that contains the font.
font_ascent_return	Returns the font ascent.
font_descent_return	Returns the font descent.
font_struct	Specifies a pointer to the *XFontStruct* structure.
nchars	Specifies the number of characters in the character string.
string	Specifies the character string.

overall_return Returns the overall size in the specified *XCharStruct* structure.

DESCRIPTION

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.

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.

XQueryTextExtents and *XQueryTextExtents16* can generate *BadFont* and *BadGC* errors.

DIAGNOSTICS

BadFont A value for a Font or GContext argument does not name a defined Font.

BadGC A value for a GContext argument does not name a defined GContext.

SEE ALSO

XTextWidth(3X11)

NAME

XTextWidth, XTextWidth16 - compute text width

SYNOPSIS

```
int XTextWidth(font_struct, string, count)
      XFontStruct *font_struct;
      char *string;
      int count;

int XTextWidth16(font_struct, string, count)
      XFontStruct *font_struct;
      XChar2b *string;
      int count;
```

ARGUMENTS

count	Specifies the character count in the specified string.
font_struct	Specifies the font used for the width computation.
string	Specifies the character string.

DESCRIPTION

The *XTextWidth* and *XTextWidth16* functions return the width of the specified 8-bit or 2-byte character strings.

SEE ALSO

XTextExtents(3X11)

NAME

XTranslateCoordinates - translate window coordinates

SYNOPSIS

```
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;
```

ARGUMENTS

child_return	Returns the child if the coordinates are contained in a mapped child of the destination window.
dest_w	Specifies the destination window.
dest_x_return *dest_y_return*	Return the x and y coordinates within the destination window.
display	Specifies the connection to the X server.
src_w	Specifies the source window.
src_x *src_y*	Specify the x and y coordinates within the source window.

DESCRIPTION

The *XTranslateCoordinates* function 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 zero, 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.

DIAGNOSTICS

BadWindow	A value for a Window argument does not name a defined Window.

NAME

XrmUniqueQuark, XrmStringToQuark, XrmQuarkToString, XrmStringToQuarkList, XrmStringToBindingQuarkList - manipulate resource quarks

SYNOPSIS

```
XrmQuark XrmUniqueQuark()

#define XrmStringToName(string) XrmStringToQuark(string) #define
XrmStringToClass(string) XrmStringToQuark(string) #define
XrmStringToRepresentation(string) XrmStringToQuark(string)

XrmQuark XrmStringToQuark(string)
     char *string;

#define XrmNameToString(name) XrmQuarkToString(name) #define
XrmClassToString(class) XrmQuarkToString(class) #define XrmRepresentationToString(type)
XrmQuarkToString(type)

char *XrmQuarkToString(quark)
     XrmQuark quark;

#define XrmStringToNameList(str, name)  XrmStringToQuarkList((str), (name)) #define
XrmStringToClassList(str,class) XrmStringToQuarkList((str), (class))

void XrmStringToQuarkList(string, quarks_return)
     char *string;
     XrmQuarkList quarks_return;

XrmStringToBindingQuarkList(string, bindings_return, quarks_return)
     char *string;
     XrmBindingList bindings_return;
     XrmQuarkList quarks_return;
```

ARGUMENTS

bindings_return	Returns the binding list.
quark	Specifies the quark for which the equivalent string is desired.
quarks_return	Returns the list of quarks.
string	Specifies the string for which a quark is to be allocated.

DESCRIPTION

The *XrmUniqueQuark* function allocates a quark that is guaranteed not to represent any string that is known to the resource manager.

These functions can be used to convert to and from quark representations. The string pointed to by the return value must not be modified or freed. If no string exists for that quark, *XrmQuarkToString* returns NULL.

The *XrmQuarkToString* function converts the specified resource quark representation back to a string.

The *XrmStringToQuarkList* function converts the null-terminated string (generally a fully qualified name) to a list of quarks. The components of the string are separated by a period or asterisk character.

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.

The *XrmStringToBindingQuarkList* function converts the specified string to a binding list and a quark list. Component names in the list are separated by a period or an asterisk character. If the string does not start with period or asterisk, a period is assumed. For example, "*a.b*c" becomes:

quarks	a	b	c
bindings	loose	tight	loose

SEE ALSO

XrmGetResource(3X11), XrmInitialize(3X11), XrmMergeDatabases(3X11), XrmPutResource(3X11)

NAME

XUnmapWindow, XUnmapSubwindows - unmap windows

SYNOPSIS

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

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

ARGUMENTS

display	Specifies the connection to the X server.
w	Specifies the window.

DESCRIPTION

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.

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.

DIAGNOSTICS

BadWindow	A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(3X11), XConfigureWindow(3X11), XCreateWindow(3X11), XDestroyWindow(3X11), XMapWindow(3X11) XRaiseWindow(3X11)

NAME

XWarpPointer - move pointer

SYNOPSIS

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;

ARGUMENTS

dest_w Specifies the destination window or *None.*

dest_x
dest_y Specify the x and y coordinates within the destination window.

display Specifies the connection to the X server.

src_x
src_y
src_width
src_height Specify a rectangle in the source window.

src_w Specifies the source window or *None*

DESCRIPTION

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 specified rectangle 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.

DIAGNOSTICS

BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XSetInputFocus(3X11)

Glossary J

Access control list

X maintains a list of hosts from which client programs can be run. By default, only programs on the local host and hosts specified in an initial list read by the server can use the display. This access control list can be changed by clients on the local host. Some servers can add or replace this mechanism with other authorization devices. The action of this mechanism can be conditional based on the authorization protocol name and data received by the server at connection setup.

Active grab

A grab is active when the pointer or keyboard is actually owned by the single grabbing client.

Ancestors

If W is an inferior of A, then A is an ancestor of W.

Atom

An atom is a unique ID corresponding to a string name. Atoms are used to identify properties, types, and selections.

Background

An `InputOutput` window can have a background, which is defined as a pixmap. When regions of the window have their contents lost or invalidated, the server automatically tiles those regions with the background.

Backing store

When a server maintains the contents of a window, the pixels saved off-screen are known as a backing store.

Bit gravity

When a window is resized, the contents of the window are not necessarily discarded. It is possible to request that the server relocate the previous contents to some region of the window (though no guarantees are made). This attraction of window contents for some location of a window is known as bit gravity.

Bit plane

When a pixmap or window is thought of as a stack of bitmaps, each bitmap is called a bit plane or plane.

Bitmap

A bitmap is a pixmap of depth one.

Border

An `InputOutput` window can have a border of equal thickness on all four sides of the window. The contents of the border are defined by a pixmap, and the server automatically maintains the contents of the border. Exposure events are never generated for border regions.

Button grabbing

Buttons on the pointer can be passively grabbed by a client. When the button is pressed, the pointer is then actively grabbed by the client.

Byte order

For image (pixmap/bitmap) data, the server defines the byte order, and clients with different native byte ordering must swap bytes as necessary. For all other parts of the protocol, the client defines the byte order, and the server swaps bytes as necessary.

Children

The children of a window are its first-level subwindows.

Class

Windows can be of different classes or types. See the entries for `InputOnly` and `InputOutput` windows for further information about valid window types.

Client

An application program connects to the window system server by some interprocess communication (IPC) path, such as a TCP connection or a shared memory buffer. This program is referred to as a client of the window system server. More precisely, the client is the IPC path itself. A program with multiple paths open to the server is viewed as multiple clients by the protocol. Resource lifetimes are controlled by connection lifetimes, not by program lifetimes.

Clipping region

In a graphics context, a bitmap or list of rectangles can be specified to restrict output to a particular region of the window. The image defined by the bitmap or rectangles is called a clipping region.

Colormap

A colormap consists of a set of entries defining color values. The colormap associated with a window is used to display the contents of the window; each pixel value indexes the colormap to produce RGB values that drive the guns of a monitor. Depending on hardware limitations, one or more colormaps can be installed at one time so that windows associated with those maps display with true colors.

Connection

The IPC path between the server and client program is known as a connection. A client program typically (but not necessarily) has one connection to the server over which requests and events are sent.

Containment

A window contains the pointer if the window is viewable and the cursor hotspot is within a visible region of the window or that of one of its inferiors. The window border is included as part of the window for containment. The pointer is in a window if the window, but no inferior, contains the pointer.

Coordinate system

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

Cursor

A cursor is the visible shape of the pointer on a screen. It consists of a hotspot, a source bitmap, a shape bitmap, and a pair of colors. The cursor defined for a window controls the visible appearance when the pointer is in that window.

Depth

The depth of a window or pixmap is the number of bits per pixel it has. The depth of a graphics context is the depth of the drawables with which it can be used.

Device

Keyboards, mice, tablets, track-balls, button boxes, and so on are all collectively known as input devices. Pointers can have one or more buttons (the most common number is three). The core protocol deals only with the keyboard and the pointer.

DirectColor

`DirectColor` is a class of colormap in which a pixel value is decomposed into three separate subfields for indexing. The first subfield indexes an array to produce red intensity values. The second subfield indexes a second array to produce blue intensity values. The third subfield indexes a third array to produce green intensity values. The RGB (red, green, and blue) values in the colormap entry can be changed dynamically.

Display

A server, together with its screens and input devices, is called a display. The Xlib `Display` structure contains all information about the particular display and its screens as well as the state that Xlib needs to communicate with the display over a particular connection.

Drawable

Both windows and pixmaps can be used as sources and destinations in graphics operations. These windows and pixmaps are collectively known as drawables. However, an `InputOnly` window cannot be used as a source or destination in a graphics operation.

Event

Clients are informed of information asynchronously by means of events. These events can be either asynchronously generated from devices or generated as side effects of client requests. Events are grouped into types. The server never sends an event to a client unless the client has specifically asked to be informed of that type of event. However, clients can force events to be sent to other clients. Events are typically reported relative to a window.

Event mask

Events are requested relative to a window. The set of event types a client requests relative to a window is described by using an event mask.

Event propagation

Device-related events propagate from the source window to ancestor windows until some client has expressed interest in handling that type of event or until the event is discarded explicitly.

Event synchronization

There are certain race conditions possible when demultiplexing device events to clients (in particular, deciding where pointer and keyboard events should be sent when in the middle of window management operations). The event synchronization mechanism allows synchronous processing of device events.

Event source

A device-related event source is the deepest viewable window that the pointer is in.

Exposure event

Servers do not guarantee to preserve the contents of windows when windows are obscured or reconfigured. Exposure events are sent to clients to inform them when contents of regions of windows have been lost.

Extension

Named extensions to the core protocol can be defined to extend the system. Extensions to output requests, resources, and event types are all possible and expected.

Font

A font is an array of glyphs (typically characters). The protocol does no translation or interpretation of character sets. The client simply indicates values used to index the glyph array. A font contains additional metric information to determine interglyph and interline spacing.

Frozen events

Clients can freeze event processing during keyboard and pointer grabs.

GC

GC is an abbreviation for graphics context. See **Graphics context.**

Glyph

A glyph is an image in a font, typically of a character.

Grab

Keyboard keys, the keyboard, pointer buttons, the pointer, and the server can be grabbed for exclusive use by a client. In general, these facilities are not intended to be used by normal applications but are intended for various input and window managers to implement various styles of user interfaces.

Graphics context

Various information for graphics output is stored in a graphics context (GC), such as foreground pixel, background pixel, line width, clipping region, and so on. A graphics context can only be used with drawables that have the same root and the same depth as the graphics context.

Gravity

Windows and window contents have a gravity that determines how the contents move when a window is resized. See **Bit gravity** and **Window gravity**.

GrayScale

`GrayScale` can be viewed as a degenerate case of `PseudoColor`, in which the red, green, and blue values in any given colormap entry are equal and thus, produce shades of gray. The gray values can be changed dynamically.

Hotspot

A cursor has an associated hotspot, which defines the point in the cursor corresponding to the coordinates reported for the pointer.

Identifier

An identifier is a unique value associated with a resource that clients use to name that resource. The identifier can be used over any connection to name the resource.

Inferiors

The inferiors of a window are all of the subwindows nested below it: the children, the children's children, and so on.

Input focus

The input focus is usually a window defining the scope for processing of keyboard input. If a generated keyboard event usually would be reported to this window or one of its inferiors, the event is reported as usual. Otherwise, the event is reported with respect to the focus window. The input focus also can be set such that all keyboard events are discarded and such that the focus window is dynamically taken to be the root window of whatever screen the pointer is on at each keyboard event.

Input manager

Control over keyboard input is typically provided by an input manager client, which usually is part of a window manager.

InputOnly window

An `InputOnly` window is a window that cannot be used for graphics requests. `InputOnly` windows are invisible and are used to control such things as cursors, input event generation, and grabbing. `InputOnly` windows cannot have `InputOutput` windows as inferiors.

InputOutput window

An `InputOutput` window is the normal kind of window that is used for both input and output. `InputOutput` windows can have both `InputOutput` and `InputOnly` windows as inferiors.

Key grabbing

Keys on the keyboard can be passively grabbed by a client. When the key is pressed, the keyboard is then actively grabbed by the client.

Keyboard grabbing

A client can actively grab control of the keyboard, and key events will be sent to that client rather than the client the events would normally have been sent to.

Keysym

An encoding of a symbol on a keycap on a keyboard.

Mapped

A window is said to be mapped if a map call has been performed on it. Unmapped windows and their inferiors are never viewable or visible.

Modifier keys

Shift, Control, Meta, Super, Hyper, Alt, Compose, Apple, CapsLock, ShiftLock, and similar keys are called modifier keys.

Monochrome

Monochrome is a special case of `StaticGray` in which there are only two colormap entries.

Obscure

A window is obscured if some other window obscures it. A window can be partially obscured and so still have visible regions. Window A obscures window B if both are viewable `InputOutput` windows, if A is higher in the global stacking order, and if the rectangle defined by the outside edges of A intersects the rectangle defined by the outside edges of B. Note the distinction between obscures and occludes. Also note that window borders are included in the calculation.

Occlude

A window is occluded if some other window occludes it. Window A occludes window B if both are mapped, if A is higher in the global stacking order, and if the rectangle defined by the outside edges of A intersects the rectangle defined by the outside edges of B. Note the distinction between occludes and obscures. Also note that window borders are included in the calculation and that `InputOnly` windows never obscure other windows but can occlude other windows.

Padding

Some padding bytes are inserted in the data stream to maintain alignment of the protocol requests on natural boundaries. This increases ease of portability to some machine architectures.

Parent window

If C is a child of P, then P is the parent of C.

Passive grab

Grabbing a key or button is a passive grab. The grab activates when the key or button is actually pressed.

Pixel value

A pixel is an N-bit value, where N is the number of bit planes used in a particular window or pixmap (that is, is the depth of the window or pixmap). A pixel in a window indexes a colormap to derive an actual color to be displayed.

Pixmap

A pixmap is a three-dimensional array of bits. A pixmap is normally thought of as a two-dimensional array of pixels, where each pixel can be a value from 0 to 2^N-1, and where N is the depth (z axis) of the pixmap. A pixmap can also be thought of as a stack of N bitmaps. A pixmap can only be used on the screen in which it was created.

Plane

When a pixmap or window is thought of as a stack of bitmaps, each bitmap is called a plane or bit plane.

Plane mask

Graphics operations can be restricted to only affect a subset of bit planes of a destination. A plane mask is a bit mask describing which planes are to be modified. The plane mask is stored in a graphics context.

Pointer

The pointing device currently attached to the cursor and tracked on the screens.

Pointer grabbing

A client can actively grab control of the pointer. Button and motion events are then sent to that client instead of the original destination client.

Pointing device

A pointing device is typically a mouse, tablet, or some other device with effective dimensional motion. The core protocol defines only one visible cursor, which tracks whatever pointing device is attached as the pointer.

Property

Windows can have associated properties that consist of a name, a type, a data format, and some data. The protocol places no interpretation on properties. They are intended as a general-purpose naming mechanism for clients. For example, clients might use properties to share information such as resize hints, program names, and icon formats with a window manager.

Property list

The property list of a window is the list of properties defined for that window.

PseudoColor

`PseudoColor` is a class of colormap in which a pixel value indexes the colormap entry to produce independent RGB values; that is, the colormap is viewed as an array of triples (RGB values). The RGB values can be changed dynamically.

Rectangle

A rectangle specified by [x,y,w,h] has an infinitely thin outline path with corners at [x,y], [x+w,y], [x+w,y+h], and [x, y+h]. When a rectangle is filled, the lower-right edges are not drawn. For example, if w=h=0, nothing would be drawn. For w=h=1, a single pixel would be drawn.

Redirecting control

Window managers (or client programs) may enforce window layout policy in various ways. When a client attempts to change the size or position of a window, the operation may be redirected to a specified client rather than the operation actually being performed.

Reply

Information requested by a client program using the X protocol is sent back to the client with a reply. Both events and replies are multiplexed on the same connection. Most requests do not generate replies, but some requests generate multiple replies.

Request

A command to the server is called a request. It is a single block of data sent over a connection.

Resource

Windows, pixmaps, cursors, fonts, graphics contexts, and colormaps are known as resources. They all have unique identifiers associated with them for naming purposes. The lifetime of a resource usually is bounded by the lifetime of the connection over which the resource was created.

RGB values

RGB values are the red, green, and blue intensity values that are used to define a color. These values are always represented as 16-bit, unsigned numbers, with 0 the minimum intensity and 65535 the maximum intensity. The X server scales these values to match the display hardware.

Root

The root of a pixmap or graphics context is the same as the root of whatever drawable was used when the pixmap or GC was created. The root of a window is the root window under which the window was created.

Root window

Each screen has a root window covering it. The root window cannot be reconfigured or unmapped, but otherwise it acts as a full-fledged window. A root window has no parent.

Save set

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 that should be remapped if currently unmapped. Save sets are typically used by window managers to avoid lost windows if the manager should terminate abnormally.

Scanline

A scanline is a list of pixel or bit values viewed as a horizontal row (all values having the same y coordinate) of an image, with the values ordered by increasing the x coordinate.

Scanline order

An image represented in scanline order contains scanlines ordered by increasing the y coordinate.

Screen

A server can provide several independent screens, which typically have physically independent monitors. This would be the expected configuration when there is only a single keyboard and pointer shared among the screens. A `Screen` structure contains the information about that screen and is linked to the `Display` structure.

Selection

A selection can be thought of as an indirect property with dynamic type. That is, rather than having the property stored in the X server, it is maintained by some client (the owner). A selection is global and is thought of as belonging to the user and being maintained by clients, rather than being private to a particular window subhierarchy or a particular set of clients. When a client asks for the contents of a selection, it specifies a selection target type, which 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, rather than the text of the paragraph. The target type can also be used for other purposes. The protocol does not constrain the semantics.

Server

The server, which is also referred to as the X server, provides the basic windowing mechanism. It handles IPC connections from clients, demultiplexes graphics requests onto the screens, and multiplexes input back to the appropriate clients.

Server grabbing

The server can be grabbed by a single client for exclusive use. This prevents processing of any requests from other client connections until the grab is completed. This is typically only a transient state for such things as rubber-banding, pop-up menus, or executing requests indivisibly.

Sibling

Children of the same parent window are known as sibling windows.

Stacking order

Sibling windows, similar to sheets of paper on a desk, can stack on top of each other. Windows above both obscure and occlude lower windows. The relationship between sibling windows is known as the stacking order.

StaticColor

`StaticColor` can be viewed as a degenerate case of `PseudoColor` in which the RGB values are predefined and read-only.

StaticGray

`StaticGray` can be viewed as a degenerate case of `GrayScale` in which the gray values are predefined and read-only. The values are typically linear or near-linear increasing ramps.

Status

Many Xlib functions return a success status. If the function does not succeed, however, its arguments are not disturbed.

Stipple

A stipple pattern is a bitmap that is used to tile a region to serve as an additional clip mask for a fill operation with the foreground color.

Tile

A pixmap can be replicated in two dimensions to tile a region. The pixmap itself is also known as a tile.

Timestamp

A timestamp is a time value expressed in milliseconds. It is typically 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, interprets timestamps from clients by treating half of the timestamp space as being earlier in time than T and half of the timestamp space as being later in time than T. One timestamp value, represented by the constant `CurrentTime`, is never generated by the server. This value is reserved for use in requests to represent the current server time.

TrueColor

`TrueColor` can be viewed as a degenerate case of `DirectColor` in which the subfields in the pixel value directly encode the corresponding RGB values. That is, the colormap has predefined read-only RGB values. The values are typically linear or near-linear increasing ramps.

Type

A type is an arbitrary atom used to identify the interpretation of property data. Types are completely uninterpreted by the server. They are solely for the benefit of clients. X predefines type atoms for many frequently used types, and clients also can define new types.

Viewable

A window is viewable if it and all of its ancestors are mapped. This does not imply that any portion of the window is actually visible. Graphics requests can be performed on a window when it is not viewable, but output will not be retained unless the server is maintaining backing store.

Visible

A region of a window is visible if someone looking at the screen can actually see it; that is, the window is viewable and the region is not occluded by any other window.

Window gravity

When windows are resized, subwindows may be repositioned automatically relative to some position in the window. This attraction of a subwindow to some part of its parent is known as window gravity.

Window manager

Manipulation of windows on the screen and much of the user interface (policy) is typically provided by a window manager client.

XY format

The data for a pixmap is said to be in XY format if it is organized as a set of bitmaps representing individual bit planes with the planes appearing from most-significant to least-significant bit order.

Z format

The data for a pixmap is said to be in Z format if it is organized as a set of pixel values in scanline order.

Index

C

D

E

F

G

H

I

J

K

L

O

P

Q

R

S

W

X

Y

Z

HEWLETT
PACKARD

HP Part Number
98794-90002

Microfiche No. 98794-99002
Printed in U.S.A.   E0989

98794-90605
For Internal Use Only