Poly/ML for X
Reference Manual

Mike Crawley

Contents

Chapter 1

Introduction

1.1 The ML interface

We have implemented an ML interface to Xlib, the industry standard C interface for X at the lowest level, and which is widely used as the basis for many toolkits. We provide all the major function groups, so that this interface can be used to implement fully functional complex applications. We also provide a set of geometric functions for handling points and rectangles, and a set of functions for performing logical operations on plane masks and pixel values.

Xlib is now widely documented, with many good reference and programming manuals available. We provide our version of the Xlib reference manual with ML signatures and types, and a more functional style to the programming interface.

We provide ML example programs to show the functionality of the ML interface to Xlib. These examples range from simple line drawing applications through to colour examples and a mini text editor showing how to program with selections. The full signatures of the structures are also provided so that modules may be written for separate compilation.

Because of the great similarity between our interface and the original Xlib, experienced X programmers can use the skills they have already developed with very few changes.

1.2 Naming and calling conventions

We have kept to the Xlib naming conventions as closely as possible. This means that standard Xlib documentation can be used along with our reference manual.

Types	**Drawable**, **Cursor**
Functions	**XDrawLines**, **XSetWindowColormap**, **XLoadQueryFont**
Constructors	**FillTiled**, **JoinMiter**, **AllowExposures**
Constants	**XA_PRIMARY**, **XA_STRING**
Labels	borderWidth

Datatypes are used where possible so that arguments are strongly typed and pattern matching may be used for returned values, this is especially useful for pattern matching the different subtypes of events. Abstypes are only used for the X resource types which have no meaningful textual representation.

The functions have been made more functional. Where an Xlib function modified its arguments, this has been changed so that the function returns new, modified copies of the arguments. Where values were passed in partially filled-in structures with OR-ed bit masks, now the programmer uses constructors to make the list of values. Similarly, return values of this type are now lists of constructed values.

The majority of X applications use a single display and single screen. Typically, they connect to the display when initialising and then pass the display parameter into every Xlib call from then on. In release 1 we connect to the display when initialising and implicitly pass the display and screen parameters to all Xlib functions. This reduces the number of parameters that have to be supplied and simplifies the signature. Another way of looking at this is to say that we have already called `XOpenDisplay` for the user and have partially applied all the Xlib functions with that display.

In subsequent releases every X resource value will have its display parameter implicitly built in, a display connection function will be provided, and the types of the other Xlib functions will be unchanged.

1.3 Event Handling

We provide an alternative event handling scheme.

In normal Xlib programs written in C the user calls XNextEvent and then has to work out which window the event is for. This soon gets unwieldy as the number of windows increases, and is very difficult to use when interfacing with toolkits of window functions. In many X toolkits each newly created window registers a function with an event handler, then events for that window are passed directly to the window function when the event reaches the head of the event queue.

We implement a similar scheme. When a window is created it is initially unhandled. It can be used for drawing on, but it will not process any events. An ML function can then be registered for that window, and an initial value supplied. The registered function will transform the value to a new value every time an event arrives for that window. In other words, a functional state machine is set up for each window. We also implement strongly typed message passing between windows, and extra event types for decoding multi-click events such as double clicking, and for implementing millisecond-resolution timer events.

In more detail, we have a single Poly/ML process that handles events arriving down the event queue. It reads each event in turn, finds the window, the window function and the window state, and applies the event and state to the function. This returns a new state, which replaces the original state. Because only one process handles the events, we guarantee that no other window function can run at the same time. Any messages 'sent' by this window function are queued up and processed when this event has finished, before the next event from the server. If the window function raises an exception, instead of returning a new state, then the current state is left unchanged, and the exception is reported at the terminal. In this way all events are handled in turn in a predictable order, and in the same way that C event handlers work. The Poly/ML top level shell process is still available for debugging and control.

If a window has an operation that takes a long time to complete, then the programmer can use Poly/ML processes to do the computations 'in the background' and 'send' the result as a message to the window for display. However, the use of processes in this way is discouraged as they are not standard.

If a window function loops, then all other windows will freeze. Since the Poly/ML top level shell is available the user can type `^C` followed by 'f' to raise Interrupt in that window function.

1.4 X and the Garbage Collector

The garbage collector in Poly/ML can detect when a value is no longer referenced, and can perform an action in this circumstance.

This is already done with streams. If you create an instream or an outstream and forget to close it with close_in or close_out, then it would hang on to its Unix file descriptor for the rest of the session. File descriptors are considered a precious resource in Unix, you are only allowed to have a small number of files open at any one time, so the garbage collector detects out of scope streams and closes the associated file descriptor.

In X there are several types of precious resources. These include Windows, GCs, Pixmaps, Fonts and Cursors. Functions are provided so that the user can explicitly reclaim the resources used by these types of object, but a similar problem occurs. If a resource is not explicitly reclaimed, and allowed to go out of scope then it can never be reclaimed by the user. The garbage collector steps in and automatically cleans up. The table below summarises the effect.

Window	close the window with **XDestroyWindow**
GC	free the **GC** with **XFreeGC**
Pixmap	free the Pixmap with **XFreePixmap**
Font	unload the **Font** with **XUnloadFont**
Cursor	free the **Cursor** with **XFreeCursor**
XColor	free the **XColor** with **XFreeColors**
Colormap	free the **Colormap** with **XFreeColormap**

Xlib includes a function called XFree which is used to free the storage required by the return types of several of its functions. This is not required in the ML interface because the garbage collector performs this operation.

1.5 X and Persistent Store

In Poly/ML, persistent store is used to carry all ML values across to the next ML session with no change, except for a couple of cases.

If you have a stream open when you save your environment, and then you attempt to read or write that stream in the next session, then Poly/ML will raise the Io exception.

In X, many values such as Windows and Pixmaps are volatile, they can only be used in the session that created them. If you attempt to draw in a window that you have brought across from an earlier session then Poly/ML will raise an exception. To make things cleaner we provide the following functions on volatile objects.

```
val DrawableExists: Drawable -> bool ;
val GCExists:       GC       -> bool ;
val FontExists:     Font     -> bool ;
val CursorExists:   Cursor   -> bool ;
val PixelExists:    int      -> bool ;
```

and so on.

These are useful for restarting applications. If an application loads fonts, generates some bitmaps, and creates some windows to work in, and then gets saved to persistent store, then when the next session is started the application can detect that its resources have evaporated and can recreate them only when needed.

Chapter 2

Function Reference

2.1 Colours, Pixels and RGB values

2.1.1 And, Or, Xor, Not, >>, <<

Types:

```
infix And Or Xor >> <<

val Not: int -> int
val And: int * int -> int
val Or:  int * int -> int
val Xor: int * int -> int
val >> : int * int -> int
val << : int * int -> int
```

Description:

These functions provide useful arithmetic operations on ints representing pixel values and plane masks, which, in X, are unsigned 32-bit quantities. The least significant bits of these quantities are on the right, and the most significant bits are on the left.

And, **Or** and **Xor** perform bitwise boolean functions.

Not performs bitwise negation, so `Not 0 = 4294967295` .

`a >> b` returns a shifted b bits to the right, where b is not negative. `a << b` returns a shifted b bits to the left, where b is not negative.

If negative values, or values greater than 4294967295 are passed to these functions then exception Range is raised.

2.1.2 BlackPixel, WhitePixel

Types:

```
val BlackPixel: unit -> int
val WhitePixel: unit -> int
```

Syntax:

```
val black = BlackPixel() ;
val white = WhitePixel() ;
```

Description:

The **BlackPixel** function returns the black pixel value for the screen.

The **WhitePixel** function returns the white pixel value for the screen.

2.1.3 Pixel, RGB, XColor

Types:

```
val Pixel: XColor -> int
val RGB:   XColor -> (int * int * int)
```

Syntax:

```
val pixel = Pixel colour ;
val (red,green,blue) = RGB colour ;
```

Arguments:

pixel	Returns the pixel field of the **XColor** structure
red	Returns the red, green and blue components of the **XColor** structure as numbers in the range 0..65535.

Argument Type:

```
datatype XColor = XColor of { doRed:   bool,
                              doGreen: bool,
                              doBlue:  bool,
                              red:     int,
                              green:   int,
                              blue:    int,
                              pixel:   int }
```

Description:

The red, green, and blue values are scaled between 0 and 65535. Full brightness in a colour is a value of 65535 independent of the number of bits actually used in the display hardware. Half brightness in a colour is a value of 32767, and off is 0. This representation gives uniform results for colour values across different screens. In some functions, the doRed, doGreen and doBlue fields control which of the red, green, and blue members are used.

2.1.4 XAllocColor, XAllocColorCells, XAllocColorPlanes, XAllocNamedColor, XFreeColors

Types:

```
val XAllocColor:       Colormap -> XColor -> XColor
val XAllocNamedColor:  Colormap -> string -> (XColor * XColor)
val XFreeColors:       Colormap -> int list -> int -> unit

val XAllocColorCells:  Colormap -> bool ->
                       int -> int -> (int list * int list)

val XAllocColorPlanes: Colormap -> bool ->
                       int -> int ->
                       int -> int -> (int list * int * int * int)
```

Syntax:

```
val real = XAllocColor cmap colour ;

val (real,desired) = XAllocNamedColor cmap name ;

XFreeColors cmap pixels planes ;

val (masks,basePixels) = XAllocColorCells cmap contig nplanes ncolours ;

val (basePixels,
     redMask,
     greenMask,
     blueMask) = XAllocColorPlanes cmap contig ncolours
                                   nreds ngreens nblues ;
```

Arguments:

name	Specifies the colour name string (for example, red) whose colour definition structure you want returned.
cmap	Specifies the colormap.
contig	Specifies a bool that indicates whether the planes must be contiguous.
ncolours	Specifies the number of pixel values that are to be returned.
nplanes	Specifies the number of plane masks that are to be returned.
nreds	Specifies the number of red planes. The value you pass must be nonnegative.
ngreens	Specifies the number of green planes. The value you pass must be nonnegative.
nblues	Specifies the number of blue planes. The value you pass must be nonnegative.
pixels	Specifies a list of pixel values.
planes	Specifies the planes you want to free.
colour	Specifies the values actually used in the colormap.
desired	Returns the exact RGB values.

real	Returns the closest RGB values provided by the hardware.
masks	Returns the list of plane masks
basePixels	Returns the list of base pixels
redMask	Returns the red mask
greenMask	Returns the green mask
blueMask	Returns the blue mask

Argument Type:

```
datatype XColor = XColor of { doRed:   bool,
                              doGreen: bool,
                              doBlue:  bool,
                              red:     int,
                              green:   int,
                              blue:    int,
                              pixel:   int }
```

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 colour closest to the specified RGB elements supported by the hardware and returns the RGB values actually used. The corresponding colormap cell is read-only. If **XAllocColor** fails then exception **XWindows** is raised with "XAllocColor failed" . Multiple clients that request the same effective RGB values can be assigned the same read-only entry, thus, allowing entries to be shared. When the last client deallocates a shared cell, it is deallocated. **XAllocColor** does not use or affect the flags in the **XColor** structure.

The **XAllocNamedColor** function looks up the named colour with respect to the screen that is associated with the specified colormap. It returns both the exact database definition and the closest colour supported by the screen. The allocated colour cell is read-only. You should use the ISO Latin-1 encoding; uppercase and lowercase do not matter. If **XAllocNamedColor** fails then exception **XWindows** is raised with "XAllocNamedColor failed" .

The **XAllocColorCells** function allocates read/write colour cells. The number of colours must be positive and the number of planes nonnegative, otherwise exception Range is raised or a **BadValue** error results. If `ncolours` and `nplanes` are requested, then `ncolours` pixels and `nplanes` 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, `ncolours * 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. If **XAllocColorCells** fails then exception **XWindows** is raised with "XAllocColorCells failed" .

The **XAllocColorPlanes** function allocates read/write colour cells. The specified ncolours must be positive; and nreds, ngreens, and nblues must be nonnegative, otherwise exception Range is raised or a **BadValue** error results. If ncolours colours, nreds reds, ngreens greens, and nblues blues are requested, ncolours 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, `ncolours * 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 `ncolours * 2 ^ nreds` independent red entries, `ncolours * 2 ^ ngreens` independent green entries, and `ncolours * 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. If **XAllocColorPlanes** fails then exception **XWindows** is raised with "XAllocColorPlanes failed" .

The **XFreeColors** function frees the cells represented by pixels whose values are in the pixels list. The planes argument should not have any bits set to 1 in common with any of the pixels. The set of all pixels is produced by ORing together subsets of the planes argument with the pixels. The request frees all of these pixels that were allocated by the client (using **XAllocColor**, **XAllocNamedColor**, **XAllocColorCells**, and **XAllocColorPlanes**). Note that freeing an individual pixel obtained from **XAllocColorPlanes** may not actually allow it to be reused until all of its related pixels are also freed. Similarly, a read-only entry is not actually freed until it has been freed by all clients, and if a client allocates the same read-only entry multiple times, it must free the entry that many times before the entry is actually freed.

All specified pixels that are allocated by the client in the colormap are freed, even if one or more pixels produce an error. If a specified pixel is not a valid index into the colormap, a **BadValue** error results. If a specified pixel is not allocated by the client (that is, is unallocated or is only allocated by another client), a **BadAccess** error results. If more than one pixel is in error, the one that gets reported is arbitrary.

2.1.5 XLookupColor, XQueryColor, XQueryColors

Types:

```
val XLookupColor: Colormap -> string -> (XColor * XColor)
val XQueryColor:  Colormap -> int -> XColor
val XQueryColors: Colormap -> int list -> XColor list
```

Syntax:

```
val (desired,real) = XLookupColor cmap name ;

val colour  = XQueryColor  cmap pixel ;
val colours = XQueryColors cmap pixels ;
```

Arguments:

colormap	Specifies the colormap.
name	Specifies the colour name string (for example, red) whose colour definition structure you want returned.
colour	Returns the RGB values for the specified pixel.
colours	Returns a list of colour definition structures for the pixel specified.
desired	Returns the exact RGB values.

real Returns the closest RGB values provided by the hardware.

Description:

The **XLookupColor** function looks up the string name of a colour with respect to the screen associated with the specified colormap. It returns both the exact colour 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** raises exception **XWindows** with "XLookupColor failed" if the name did not exist.

The **XQueryColor** function returns the hardware-specific RGB values for the specified pixel and sets the DoRed, DoGreen, and DoBlue flags. The **XQueryColors** function returns the RGB values for each pixel in the list and sets the DoRed, DoGreen, and DoBlue flags.

2.1.6 XParseColor

Types:

```
val XParseColor: Colormap -> string -> XColor
```

Syntax:

```
val colour = XParseColor cmap name ;
```

Arguments:

cmap Specifies the colormap.

name Specifies the colour name string; case is ignored.

colour Returns the exact colour value for later use and sets the doRed, doGreen, and doBlue flags.

Argument Type:

```
datatype XColor = XColor of { doRed:   bool,
                              doGreen: bool,
                              doBlue:  bool,
                              red:     int,
                              green:   int,
                              blue:    int,
                              pixel:   int }
```

Description:

The **XParseColor** function provides a simple way to create a standard user interface to colour. It takes a string specification of a colour, 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 colour can be specified either as a colour 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 colour 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 colour does not exist in the server's database, then exception **XWindows** is raised with "XParseColor failed" .

2.1.7 XStoreColor, XStoreColors, XStoreNamedColor

Types:

```
val XStoreColor:      Colormap -> XColor -> unit
val XStoreColors:     Colormap -> XColor list -> unit
val XStoreNamedColor: Colormap -> string -> int -> (bool * bool * bool) -> unit
```

Syntax:

```
XStoreColor cmap colour ;
XStoreColors cmap colours ;
XStoreNamedColor cmap name pixel (doRed,doGreen,doBlue) ;
```

Arguments:

colour	Specifies the pixel and RGB values
colours	Specifies a list of pixel and RGB values
cmap	Specifies the colormap.
doRed	Specifies if the red component is set
doGreen	Specifies if the green component is set
doBlue	Specifies if the blue component is set.
name	Name of colour to copy RGB values from.
pixel	Specifies the entry in the colormap.

Argument Type:

```
datatype XColor = XColor of { doRed:   bool,
                              doGreen: bool,
                              doBlue:  bool,
                              red:     int,
                              green:   int,
                              blue:    int,
                              pixel:   int }
```

Description:

The **XStoreColors** function changes the colormap entries of the pixel values specified in the pixel members of the **XColor** structures. You specify which colour components are to be changed by setting doRed, doGreen, and/or doBlue in 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.

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 colour components. You specify which colour components are to be changed by setting doRed, doGreen, and/or doBlue in the **XColor** structure. If the colormap is an installed map for its screen, the changes are visible immediately.

The **XStoreNamedColor** function looks up the named colour 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 booleans doRed, doGreen, and doBlue determine which of the red, green, and blue components are set. 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.

2.2 Colormaps

2.2.1 DefaultColormap

Types:

```
val DefaultColormap: unit -> Colormap
```

Syntax:

```
val cmap = DefaultColormap() ;
```

Description:

The **DefaultColormap** function returns the default colormap for allocation on the screen.

2.2.2 DefaultDepth

Types:

```
val DefaultDepth: unit -> int
```

Syntax:

```
val depth = DefaultDepth() ;
```

Description:

The **DefaultDepth** function returns the depth (number of planes) of the default root window for the screen.

2.2.3 DisplayCells

Types:

```
val DisplayCells: unit -> int
```

Syntax:

```
val cells = DisplayCells() ;
```

Description:

The **DisplayCells** function returns the number of entries in the default colormap.

2.2.4 VisualClass

Types:

```
val VisualClass: Visual -> VisualClass
```

Syntax:

```
val class = VisualClass visual ;
```

Arguments:

visual	Specifies the visual.
class	Returns the class from the visual.

Argument Type:

```
datatype VisualClass = StaticGray  | GrayScale
                     | StaticColor | PseudoColor
                     | TrueColor   | DirectColor
```

Description:

Returns the visual class from the specified visual.

2.2.5 XCreateColormap, XCopyColormapAndFree, XFreeColormap, XSetWindowColormap

Types:

```
val XCreateColormap:      Drawable -> Visual -> AllocType -> Colormap
val XCopyColormapAndFree: Colormap -> Colormap
val XFreeColormap:        Colormap -> unit
val XSetWindowColormap:   Drawable -> Colormap -> unit
```

Syntax:

```
val cmap = XCreateColormap w visual alloc ;
val copy = XCopyColormapAndFree cmap ;
XFreeColormap cmap ;
XSetWindowColormap w cmap ;
```

Arguments:

w	Specifies the window
visual	Specifies a visual type supported on the screen. If the visual type is not one supported by the screen, a **BadMatch** error results.
alloc	Specifies the colormap entries to be allocated. You can pass **AllocNone** or **AllocAll**.
cmap	Specifies the colormap.
copy	Returns a copy of the colormap.

Argument Type:

```
datatype AllocType = AllocNone | AllocAll

datatype VisualClass = StaticGray  | GrayScale
                     | StaticColor | PseudoColor
                     | TrueColor   | DirectColor

datatype XColor = XColor of { doRed:   bool,
                              doGreen: bool,
                              doBlue:  bool,
                              red:     int,
                              green:   int,
                              blue:    int,
                              pixel:   int }
```

Argument Description:

The red, green, and blue values are scaled between 0 and 65535. Full brightness in a colour is a value of 65535 independent of the number of bits actually used in the display hardware. Half brightness in a colour is a value of 32767, and off is 0. This representation gives uniform results for colour values across different screens. In some functions, the doRed, doGreen and doBlue fields control which of the red, green, and blue members are used.

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 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.

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 redMask, greenMask, and blueMask 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**.

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. It also moves all of the client's existing allocation from the specified colormap to the new colormap with their colour 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 colour 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.

The **XFreeColormap** function deletes the association between the colormap resource in the server and the ML **Colormap** value. 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 **NoColormap** and generates a **ColormapNotify** event. X does not define the colours displayed for a window with a colormap of **NoColormap**.

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.

2.2.6 XInstallColormap, XUninstallColormap, XListInstalledColormaps

Types:

```
val XInstallColormap:        Colormap -> unit
val XListInstalledColormaps: Drawable -> Colormap list
val XUninstallColormap:      Colormap -> unit
```

Syntax:

```
XInstallColormap cmap ;
XUninstallColormap cmap ;
val cmaps = XListInstalledColormaps w ;
```

Arguments:

cmap	Specifies the colormap.
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 colours. 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.

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.

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. If **XListInstalledColormaps** fails then exception **XWindows** is raised with "XListInstalledColormaps failed" .

2.2.7 XSetRGBColormaps, XGetRGBColormaps

Types:

```
val XSetRGBColormaps: Drawable -> int -> XStandardColormap list -> unit
val XGetRGBColormaps: Drawable -> int -> XStandardColormap list
```

Syntax:

```
XSetRGBColormaps w property stdmaps ;
val maps = XGetRGBColormaps w property ;
```

Arguments:

w	Specifies the window.

property	Specifies the property atom.
stdmaps	Specifies the XStandardColormaps to be used
maps	Returns the **XStandardColormap**

Argument Type:

```
datatype XStandardColormap = XStandardColormap of { colormap:  Colormap,
                                                    redMax:    int,
                                                    redMult:   int,
                                                    greenMax:  int,
                                                    greenMult: int,
                                                    blueMax:   int,
                                                    blueMult:  int,
                                                    basePixel: int,
                                                    visual:    Visual }
```

Argument Description:

The colormap member is the colormap created by the **XCreateColormap** function. The redMax, greenMax, and blueMax members give the maximum red, green, and blue values, respectively. Each colour 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 redMax = 7, greenMax = 7, and blueMax = 3. An alternate allocation that uses only 216 colours is redMax = 5, greenMax = 5, and blueMax = 5.

The redMult, greenMult, and blueMult members give the scale factors used to compose a full pixel value. (See the discussion of the basePixel members for further information.) For a 3/3/2 allocation, redMult might be 32, greenMult might be 4, and blueMult might be 1. For a 6-colours-each allocation, redMult might be 36, greenMult might be 6, and blueMult might be 1.

The basePixel member gives the base pixel value used to compose a full pixel value. Usually, the basePixel 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 * redMult + g * greenMult + b * blueMult + basePixel
```

For **GrayScale** colormaps, only the colormap, redMax, redMult, and basePixel members are defined. The other members are ignored.

To compute a **GrayScale** pixel value, use the following expression:

```
gray * redMult + basePixel
```

The visual member gives the the visual from which the colormap was created.

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

Description:

XSetRGBColormaps sets the RGB colormap definition in the specified property on the named window. The property is stored with a type of **RGB_COLOR_MAP** and a format of 32. Note that it is the caller's responsibility to honour the ICCCM restriction that only **RGB_DEFAULT_MAP** can contain more than one definition.

The **XGetRGBColormaps** function returns the RGB colormap definitions stored in the specified property on the named window. If the property exists, is of type **RGB_COLOR_MAP**, is of format 32, and is long enough to contain a colormap definition (if the visual is not present, **XGetRGBColormaps** assumes the default visual for the screen on which the window is located), **XGetRGBColormaps** returns the list of colormaps. Otherwise, **XGetRGBColormaps** returns the empty list. Note that it is the caller's responsibility to honour the ICCCM restriction that only **RGB_DEFAULT_MAP** can contain more than one definition.

2.3 Cursors

2.3.1 XCreateFontCursor, XCreatePixmapCursor, XCreateGlyphCursor

Types:

```
val XCreateFontCursor:   int -> Cursor

val XCreatePixmapCursor: Drawable -> Drawable ->
                         XColor -> XColor -> XPoint -> Cursor

val XCreateGlyphCursor:  Font -> Font ->
                         int  -> int ->
                         XColor -> XColor -> Cursor
```

Syntax:

```
val cursor = XCreateFontCursor shape ;

val cursor = XCreatePixmapCursor source mask
                                 foreground background hotspot ;

val cursor = XCreateGlyphCursor sourceFont maskFont
                                 sourceChar maskChar
                                 foreground background ;
```

Arguments:

background	Specifies the RGB values for the background of the source.
foreground	Specifies the RGB values for the foreground of the source.
mask	Specifies the cursor's mask bits to be displayed or **NoDrawable**.
maskChar	Specifies the glyph character for the mask.
maskFont	Specifies the font for the mask glyph or **NoFont**.
shape	Specifies the shape name of the cursor.
source	Specifies the cursor's source bits to be displayed.
sourceChar	Specifies the character glyph for the source.
sourceFont	Specifies the font for the source glyph.

hotspot	Specifies the x and y coordinates, which indicate the hotspot relative to the source's origin.
cursor	Returns the new cursor

Argument Type:

```
datatype XColor = XColor of { doRed:   bool,
                              doGreen: bool,
                              doBlue:  bool,
                              red:     int,
                              green:   int,
                              blue:    int,
                              pixel:   int }
```

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 colours of a cursor are a black foreground and a white background (see **XRecolorCursor**).

The **XCreatePixmapCursor** function creates and returns a cursor. The foreground and background RGB values must be specified using foreground and background, even if the X server only has a **StaticGray** or **GrayScale** screen. The foreground colour is used for the pixels set to 1 in the source, and the background colour 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.

The **XCreateGlyphCursor** function is similar to **XCreatePixmapCursor** except that the source and mask bitmaps are obtained from the specified font glyphs. The sourceChar must be a defined glyph in sourceFont, or a **BadValue** error results. If maskFont is given, maskChar must be a defined glyph in maskFont, or a **BadValue** error results. The maskFont and maskChar are optional. The origins of the sourceChar and maskChar (if defined) glyphs are positioned coincidently and define the hotspot. The sourceChar and maskChar 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 maskChar 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.

2.3.2 XDefineCursor, XUndefineCursor, NoCursor

Types:

```
val XDefineCursor:   Drawable -> Cursor -> unit
val XUndefineCursor: Drawable -> unit
val NoCursor:        Cursor
```

Syntax:

```
XDefineCursor w cursor ;
XUndefineCursor w ;
```

Arguments:

cursor	Specifies the cursor that is to be displayed or **NoCursor**.
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 **NoCursor**, it is equivalent to **XUndefineCursor**.

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.

2.3.3 XRecolorCursor, XFreeCursor

Types:

```
val XRecolorCursor: Cursor -> XColor -> XColor -> unit
val XFreeCursor:    Cursor   -> unit
```

Syntax:

```
XRecolorCursor cursor fg bg ;
XFreeCursor cursor ;
```

Arguments:

bg	Specifies the RGB values for the background of the source.
cursor	Specifies the cursor.
fg	Specifies the RGB values for the foreground of the source.

Description:

The **XRecolorCursor** function changes the colour of the specified cursor, and if the cursor is being displayed on a screen, the change is visible immediately.

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

2.4 Display Specifications

2.4.1 AllPlanes

Types:

```
val AllPlanes: int
```

Syntax:

```
val planeMask = AllPlanes ;
```

Description:

AllPlanes is a value with all bits set to 1 and is suitable for use in a plane mask argument to a function.

2.4.2 BitmapBitOrder

Types:

```
val BitmapBitOrder: unit -> ImageOrder
```

Argument Type:

```
datatype ImageOrder = LSBFirst | MSBFirst
```

Syntax:

```
val order = BitmapBitOrder() ;
```

Description:

The **BitmapBitOrder** function 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 bytes comprising the bitmap data.

2.4.3 BitmapPad

Types:

```
val BitmapPad: unit -> int
```

Syntax:

```
val pad = BitmapPad() ;
```

Description:

The **BitmapPad** function returns the number of bits that each scanline must be padded.

2.4.4 BitmapUnit

Types:

```
val BitmapUnit: unit -> int
```

Syntax:

```
val scanline = BitmapUnit() ;
```

Description:

The **BitmapUnit** function returns the size of a bitmap's scanline unit in bits.

2.4.5 ByteOrder

Types:

```
val ByteOrder: unit -> ImageOrder
```

Argument Type:

```
datatype ImageOrder = LSBFirst | MSBFirst
```

Syntax:

```
val order = ByteOrder ;
```

Description:

The **ByteOrder** function specifies the required byte order for images for each scanline unit in XY format (bitmap) or for each pixel value in Z format.

2.4.6 CellsOfScreen

Types:

```
val CellsOfScreen: unit -> int
```

Syntax:

```
val cells = CellsOfScreen() ;
```

Description:

The **CellsOfScreen** function returns the number of colormap cells in the default colormap of the screen.

2.4.7 ColormapExists, CursorExists, DrawableExists, FontExists, GCExists, VisualExists

Types:

```
val ColormapExists: Colormap -> bool
val CursorExists:   Cursor   -> bool
val DrawableExists: Drawable -> bool
val FontExists:     Font     -> bool
val GCExists:       GC       -> bool
val VisualExists:   Visual   -> bool
```

Description:

In Poly/ML all values may be committed to the database and then referenced in future Poly/ML sessions. X resources are stored in the X server and are destroyed at the end of every Poly/ML session. If the user attempts to use an ML value corresponding to an X resource that existed in an earlier session, then exception **XWindows** is raised with "Non-existant resource" . To allow programmers to detect old resources these functions return true only if the ML value passed in corresponds to an X resource created in this session, and return false otherwise.

2.4.8 ColormapID, CursorID, DrawableID, FontID, GCID, VisualID, SameDrawable

Types:

```
type Colormap ;
type Cursor ;
type Drawable ;
type Font ;
type GC ;
type Visual ;

val ColormapID: Colormap -> int
val CursorID:   Cursor   -> int
val DrawableID: Drawable -> int
val FontID:     Font     -> int
val GCID:       GC       -> int
val VisualID:   Visual   -> int

val SameDrawable: Drawable -> Drawable -> bool
```

Description:

These functions return the X identifiers for the corresponding ML value. In X, unique numbers are generated for client resources such as windows and pixmaps, and these numbers are sent in the messages between the X server and the client to identify the resources.

If two resources have the same X identifier, then they are the same resource. Thus the convenience function **SameDrawable** is defined as:

```
fun SameDrawable a b = (DrawableID a = DrawableID b)
```

2.4.9 DefaultVisual

Types:

```
val DefaultVisual: unit -> Visual
```

Syntax:

```
val visual = DefaultVisual() ;
```

Description:

The **DefaultVisual** function returns the default visual type for the screen.

2.4.10 DisplayConnected

Types:

```
val DisplayConnected: unit -> bool
```

Description:

In release 1 of the X Window interface in Poly/ML, the display is connected to automatically when Poly/ML starts. If `-noDisplay` was specified on the command line, or Poly/ML cannot connect to the display for whatever reason, then Poly/ML runs without a display connected. An attempt to use an X function that needs the display will raise exception **XWindows** with "Display not connected" . To allow programmers to avoid this situation, this function returns true only if the display is connected, and false otherwise.

2.4.11 DisplayHeight, DisplayHeightMM, DisplayWidth, DisplayWidthMM

Types:

```
val DisplayHeight:   unit -> int
val DisplayHeightMM: unit -> int
val DisplayWidth:    unit -> int
val DisplayWidthMM:  unit -> int
```

Syntax:

```
val height = DisplayHeight() ;
val height = DisplayHeightMM() ;
val width  = DisplayWidth() ;
val width  = DisplayWidthMM() ;
```

Description:

The **DisplayHeight** function returns the height of the specified screen in pixels.

The **DisplayHeightMM** function returns the height of the screen in millimeters.

The **DisplayWidth** function returns the width of the screen in pixels.

The **DisplayWidthMM** function returns the width of the specified screen in millimeters.

2.4.12 DisplayPlanes

Types:

```
val DisplayPlanes: unit -> int
```

Syntax:

```
val planes = DisplayPlanes() ;
```

Description:

The **DisplayPlanes** function returns the depth of the root window of the screen.

2.4.13 DisplayString

Types:

```
val DisplayString: unit -> string
```

Syntax:

```
val s = DisplayString() ;
```

Description:

The **DisplayString** function returns the string that was passed to XOpenDisplay when the current display was opened.

2.4.14 DoesBackingStore

Types:

```
val DoesBackingStore: unit -> BackingStore
```

Syntax:

```
val bs = DoesBackingStore() ;
```

Argument Type:

```
datatype BackingStore = NotUseful | WhenMapped | Always
```

Description:

The **DoesBackingStore** function returns **WhenMapped**, **NotUseful**, or **Always**, which indicate whether the screen supports backing stores.

2.4.15 DoesSaveUnders

Types:

```
val DoesSaveUnders: unit -> bool
```

Syntax:

```
val su = DoesSaveUnders() ;
```

Description:

The **DoesSaveUnders** function returns a bool indicating whether the screen supports save unders.

2.4.16 EventMaskOfScreen

Types:

```
val EventMaskOfScreen: unit -> EventMask list
```

Syntax:

```
val mask = EventMaskOfScreen() ;
```

Description:

The **EventMaskOfScreen** function returns the root event mask of the root window for the screen at connection setup.

2.4.17 MinCmapsOfScreen

Types:

```
val MinCmapsOfScreen:  unit -> int
```

Syntax:

```
val n = MinCmapsOfScreen() ;
```

Description:

The **MinCmapsOfScreen** function returns the minimum number of installed colormaps supported by the screen.

2.4.18 MaxCmapsOfScreen

Types:

```
val MaxCmapsOfScreen:  unit -> int
```

Syntax:

```
val n = MaxCmapsOfScreen() ;
```

Description:

The **MaxCmapsOfScreen** function returns the maximum number of installed colormaps supported by the screen.

2.4.19 NoColormap, NoCursor, NoDrawable, NoFont, NoVisual, ParentRelative, CopyFromParentDrawable, CopyFromParentVisual, PointerWindow, InputFocus, PointerRoot

Types:

```
val NoColormap:             Colormap
val NoCursor:               Cursor
val NoDrawable:             Drawable
val NoFont:                 Font
val NoVisual:               Visual
val ParentRelative:         Drawable
val CopyFromParentDrawable: Drawable
val CopyFromParentVisual:   Visual
val PointerWindow:          Drawable
val InputFocus:             Drawable
val PointerRoot:            Drawable
```

Description:

These names refer to constant values of the indicated type that may be used instead of passing a real, live instance of one of these types. Typically they are used to indicate that some default action should take place. For example, setting the background pixmap of a window to **ParentRelative** specifies that the background pixmap of the window's parent is to be used.

2.4.20 ProtocolRevision

Types:

```
val ProtocolRevision: unit -> int
```

Syntax:

```
val rev = ProtocolRevision() ;
```

Description:

The **ProtocolRevision** function returns the minor protocol revision number of the X server.

2.4.21 ProtocolVersion

Types:

```
val ProtocolVersion: unit -> int
```

Syntax:

```
val v = ProtocolVersion() ;
```

Description:

The **ProtocolVersion** function returns the major version number (11) of the X protocol associated with the connected display.

2.4.22 RootWindow

Types:

```
val RootWindow: unit -> Drawable
```

Syntax:

```
val root = RootWindow() ;
```

Description:

The **RootWindow** function returns the root window.

2.4.23 ServerVendor

Types:

```
val ServerVendor: unit -> string
```

Syntax:

```
val s = ServerVendor() ;
```

Description:

The **ServerVendor** function returns a string that provides some identification of the owner of the X server implementation.

2.4.24 VendorRelease

Types:

```
val VendorRelease: unit -> int
```

Syntax:

```
val n = VendorRelease() ;
```

Description:

The **VendorRelease** function returns a number related to a vendor's release of the X server.

2.4.25 XQueryBestCursor, XQueryBestSize, XQueryBestStipple, XQueryBestTile

Types:

```
val XQueryBestSize: ShapeClass -> Drawable -> XRectangle -> XRectangle

val XQueryBestCursor:  Drawable -> XRectangle -> XRectangle
val XQueryBestStipple: Drawable -> XRectangle -> XRectangle
val XQueryBestTile:    Drawable -> XRectangle -> XRectangle
```

Syntax:

```
val bestSize = XQueryBestCursor  whichScreen area ;
val bestSize = XQueryBestSize    whichScreen class area ;
val bestSize = XQueryBestStipple whichScreen area ;
val bestSize = XQueryBestTile    whichScreen area ;
```

Argument Type:

```
datatype ShapeClass = CursorShape | TileShape | StippleShape
```

Arguments:

class	Specifies the class that you are interested in. You can pass **TileShape**, **CursorShape**, or **StippleShape**.
whichScreen	**Drawable** to determine which screen.
area	Specifies the width and height.
bestSize	Returns 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 whichScreen. 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.

The **XQueryBestTile** function returns the best or closest size, that is, the size that can be tiled fastest on the screen specified by d. 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.

The **XQueryBestStipple** function returns the best or closest size, that is, the size that can be stippled fastest on the screen specified by whichScreen. 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.

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.

2.5 Drawing Primitives

2.5.1 XClearArea, XClearWindow

Types:

```
val XClearArea:   Drawable -> XRectangle -> bool -> unit
val XClearWindow: Drawable -> unit
```

Syntax:

```
XClearArea w area exposures ;
XClearWindow w ;
```

Arguments:

exposures	Specifies a bool that indicates if **Expose** events are to be generated.
area	Specifies the area to be cleared in the window.
w	Specifies the window.

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 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 **NoDrawable**, 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 **InputOnlyClass**, a **BadMatch** error results.

The **XClearWindow** function clears the entire area in the specified window and is equivalent to `XClearArea w empty 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 **NoDrawable**, the contents of the window are not changed. If you specify a window whose class is **InputOnlyClass**, a **BadMatch** error results.

2.5.2 XCopyArea, XCopyPlane

Types:

```
val XCopyArea:  Drawable -> Drawable -> GC ->
                XPoint -> XRectangle -> unit

val XCopyPlane: Drawable -> Drawable -> GC ->
                XPoint -> XRectangle -> int -> unit
```

Syntax:

```
XCopyArea  src dest gc srcPoint destArea ;
XCopyPlane src dest gc srcPoint destArea plane ;
```

Arguments:

destArea	Specifies the destination rectangle
gc	Specifies the **GC**.
plane	Specifies the bit plane. You must set exactly one bit to 1.
src	Specifies the source
dest	and destination drawables to be combined.
srcPoint	Specifies the upper-left corner of the source rectangle.

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 **NoDrawable**, 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-origin, and clip-mask.

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 is less than `2 ^ n` , where n is the depth of the drawables, 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-origin, and clip-mask.

2.5.3 XDrawArc, XDrawArcs

Types:

```
val XDrawArc:           Drawable -> GC -> XArc -> unit
val XDrawArcs:          Drawable -> GC -> XArc list -> unit
```

Syntax:

```
XDrawArc d gc (XArc (area,angle1,angle2)) ;
XDrawArcs d gc arcs ;
```

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 list of arcs.
d	Specifies the drawable.
gc	Specifies the **GC**.
area	Specifies the bounding rectangle of the area. The x and y coordinates, which are relative to the origin of the drawable, specify the upper-left corner of the bounding rectangle. The width and height are the major and minor axes of the arc.

Argument Type:

```
datatype XArc = XArc of XRectangle * int * int
```

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

```
(XArc (Area {x,y,width,height}),angle1,angle2),
```

the origin of the major and minor axes is at

```
(x + width div 2,y + height div 2),
```

and the infinitely thin path describing the entire circle or ellipse intersects the horizontal axis at

```
(x,y + height div 2) and  (x + width,y + height div 2)
```

and intersects the vertical axis at

```
(x + width div 2,y) and (x + width div 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

```
(XArc (Area {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:

```
skewed-angle = atan (tan normal-angle * width div height) + 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 (˜pi/2,pi/2) and adjust is:

0 for normal-angle in the range (0,pi/2)

pi for normal-angle in the range (pi/2,3*pi/2)

2*pi for normal-angle in the range (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: foreground, background, tile, stipple, tile-stipple-origin, dash-offset, dash-list, function, plane-mask, line-width, line-style, cap-style, join-style, fill-style, subwindow-mode, clip-origin, and clip-mask.

2.5.4 XDrawImageString, XDrawImageString16

Types:

```
val XDrawImageString:   Drawable -> GC -> XPoint -> string   -> unit
val XDrawImageString16: Drawable -> GC -> XPoint -> int list -> unit
```

Syntax:

```
XDrawImageString d gc point string ;
XDrawImageString16 d gc point bigChars ;
```

Arguments:

d	Specifies the drawable.
gc	Specifies the **GC**.
string	Specifies the character string.
bigChars	Specifies the character string as a list of 16 bit integers.
point	Specifies 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 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-ascent), the width is overall, and the height is ascent+descent. The overall, ascent, and descent are as would be returned by **XTextExtents** using the font in the 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 16-bit matrix indexing and used with **XDrawImageString**, each 8-bit character in the string is used to form the least-significant 8-bits of the index, the most-significant bits are taken to be zero.

Both functions use these **GC** components: plane-mask, foreground, background, font, subwindow-mode, clip-origin, and clip-mask.

2.5.5 XDrawLine, XDrawLines, XDrawSegments

Types:

```
val XDrawLine:     Drawable -> GC -> XPoint -> XPoint -> unit
val XDrawLines:    Drawable -> GC -> XPoint list -> CoordMode -> unit
val XDrawSegments: Drawable -> GC -> (XPoint * XPoint) list -> unit
```

Syntax:

```
XDrawLine d gc point1 point2 ;
XDrawLines d gc points mode ;
XDrawSegments d gc segments ;
```

Arguments:

d	Specifies the drawable.
gc	Specifies the **GC**.
mode	Specifies the coordinate mode. You can pass **CoordModeOrigin** or **CoordModePrevious**.
points	Specifies a list of points.
segments	Specifies a list of pairs of points.
point1	Specifies the points
point2	to be connected.

Argument Type:

```
datatype CoordMode = CoordModeOrigin | CoordModePrevious
```

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 list of **XPoint** structures. It draws the lines in the same order as the list. 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 same order as the list of pairs of points 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: foreground, background, tile, stipple, tile-stipple-origin, dash-offset, dash-list, function, plane-mask, line-width, line-style, cap-style, fill-style, subwindow-mode, clip-origin, and clip-mask. The **XDrawLines** function also uses the join-style **GC** component.

2.5.6 XDrawPoint, XDrawPoints

Types:

```
val XDrawPoint:  Drawable -> GC -> XPoint -> unit
val XDrawPoints: Drawable -> GC -> XPoint list -> CoordMode -> unit
```

Syntax:

```
XDrawPoint d gc point ;
XDrawPoints d gc points mode ;
```

Arguments:

d	Specifies the drawable.
gc	Specifies the **GC**.
points	Specifies a list of points.
point	Specifies the point.
mode	Specifies the coordinate mode. You can pass **CoordModeOrigin** or **CoordModePrevious**.

Argument Type:

```
datatype CoordMode = CoordModeOrigin | CoordModePrevious
```

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 same order as the list.

Both functions use these **GC** components: function, plane-mask, foreground, subwindow-mode, clip-origin, and clip-mask.

2.5.7 XDrawRectangle, XDrawRectangles

Types:

```
val XDrawRectangle:     Drawable -> GC -> XRectangle -> unit
val XDrawRectangles:    Drawable -> GC -> XRectangle list -> unit
```

Syntax:

```
XDrawRectangle d gc (Area{x=x,y=y,w=width,h=height}) ;
XDrawRectangles d gc rectangles ;
```

Arguments:

d	Specifies the drawable.

gc	Specifies the **GC**.
rectangles	Specifies a list of rectangles.
x,y	Specifies the upper-left corner of the rectangle.
width	Specifies the dimensions
height	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 same order as the list. If rectangles intersect, the intersecting pixels are drawn multiple times.

Both functions use these **GC** components: foreground, background, tile, stipple, tile-stipple-origin, dash-offset, dash-list, function, plane-mask, line-width, line-style, join-style, fill-style, subwindow-mode, clip-origin, and clip-mask.

2.5.8 XDrawString, XDrawString16

Types:

```
val XDrawString:   Drawable -> GC -> XPoint -> string   -> unit
val XDrawString16: Drawable -> GC -> XPoint -> int list -> unit
```

Syntax:

```
XDrawString d gc point string ;
XDrawString16 d gc point bigChars ;
```

Arguments:

d	Specifies the drawable.
gc	Specifies the **GC**.
string	Specifies the character string.
bigChars	Specifies the character string as a list of 16-bit integers.
point	Specifies 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 with 2-byte indexing rather than 16-bit linear indexing, pass byte 1 as the most-significant 8-bits and byte 2 as the least-significant 8-bits in the bigChars argument.

For fonts defined with 16-bit linear indexing and used with **XDrawString**, each 8-bit character in the string is used to form the least-significant 8-bits of the index, the most-significant bits are taken to be zero.

For fonts defined with 2-byte matrix indexing and used with **XDrawString**, each 8-bit character in the string is used to form byte 2 of the index, byte 1 is taken to be zero.

Both functions use these **GC** components: foreground, background, tile, stipple, tile-stipple-origin, function, plane-mask, fill-style, font, subwindow-mode, clip-origin, and clip-mask.

2.5.9 XDrawText, XDrawText16

Types:

```
val XDrawText:   Drawable -> GC -> XPoint -> XTextItem list -> unit
val XDrawText16: Drawable -> GC -> XPoint -> XTextItem16 list -> unit
```

Syntax:

```
XDrawText d gc point items ;
XDrawText16 d gc point items ;
```

Arguments:

d	Specifies the drawable.
gc	Specifies the **GC**.
point	Specifies 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 list of text items.

Argument Type:

```
datatype XTextItem   = XTextItem   of string   * int * Font
datatype XTextItem16 = XTextItem16 of int list * int * Font
```

Argument Description:

If the font member is not **NoFont**, 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.

Description:

The **XDrawText16** function is similar to **XDrawText** except that it uses 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 **NoFont** 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 with 2-byte indexing rather than 16-bit linear indexing, pass byte 1 as the high order 8-bits and byte 2 as the low order 8-bits in the **XTextItem16** list.

Both functions use these **GC** components: foreground, background, tile, stipple, tile-stipple-origin, function, plane-mask, fill-style, font, subwindow-mode, clip-origin, and clip-mask.

2.5.10 XFillArc, XFillArcs, XFillPolygon, XFillRectangle, XFillRectangles

Types:

```
val XFillArc:        Drawable -> GC -> XArc -> unit
val XFillArcs:       Drawable -> GC -> XArc list -> unit
val XFillRectangle:  Drawable -> GC -> XRectangle -> unit
val XFillRectangles: Drawable -> GC -> XRectangle list -> unit

val XFillPolygon: Drawable -> GC ->
                  XPoint list -> PolyShape -> CoordMode -> unit
```

Syntax:

```
XFillArc d gc (XArc (area,angle1,angle2)) ;
XFillArcs d gc arcs ;
XFillPolygon d gc points shape mode ;
XFillRectangle d gc (Area{x=x,y=y,w=width,h=height}) ;
XFillRectangles d gc rectangles ;
```

Argument Type:

```
datatype PolyShape = Complex | Nonconvex | Convex

datatype CoordMode = CoordModeOrigin | CoordModePrevious
```

Arguments:

d	Specifies the drawable.
gc	Specifies the **GC**.

area	Specifies the bounding rectangle of the area. The x and y coordinates, which are relative to the origin of the drawable, specify the upper-left corner of the bounding rectangle. The width and height are the major and minor axes of the arc.
angle1	Specifies the start of the arc relative to the three-o'clock position from the center, in units of degrees * 64.
angle2	Specifies the path and extent of the arc relative to the start of the arc, in units of degrees * 64.
arcs	Specifies a list of arcs.
points	Specifies a list of points.
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**.
x,y	Specifies the upper-left corner of the rectangle.
width	Specifies the dimensions
height	of the rectangle.
rectangles	Specifies a list of rectangles.

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 same order as the list. 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: foreground, background, tile, stipple, tile-stipple-origin, function, plane-mask, fill-style, subwindow-mode, clip-origin, and clip-mask.

XFillPolygon fills the region closed by the specified path. The path is closed automatically if the last point in the list does not coincide with the first point. **XFillPolygon** does not draw a pixel of the region more than once. **CoordModeOrigin** treats all coordinates as relative to the origin, and **CoordModePrevious** treats all coordinates after the first as relative to the previous point.

Depending on the specified shape, the following occurs:

If shape is **Complex**, the path may self-intersect. Note that contiguous coincident points in the path are not treated as self-intersection.

If shape is **Convex**, for every pair of points inside the polygon, the line segment connecting them does not intersect the path. If known by the client, specifying **Convex** can improve performance. If you specify **Convex** for a path that is not convex, the graphics results are undefined.

If shape is **Nonconvex**, the path does not self-intersect, but the shape is not wholly convex. If known by the client, specifying **Nonconvex** instead of **Complex** may improve performance. If you specify **Nonconvex** for a self-intersecting path, the graphics results are undefined.

The fill-rule of the **GC** controls the filling behavior of self-intersecting polygons.

This function uses these **GC** components: foreground, background, tile, stipple, tile-stipple-origin, function, plane-mask, fill-style, fill-rule, subwindow-mode, clip-origin, and clip-mask.

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 same order as the list. 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: foreground, background, tile, stipple, tile-stipple-origin, function, plane-mask, fill-style, arc-mode, subwindow-mode, clip-origin, and clip-mask.

2.6 Exceptions

2.6.1 Range

Types:

```
exception Range
```

Description:

Range is raised when an argument to a function is not inside the allowable range of values. There are many restricted ranges for function arguments. In brief:

x and y coordinates must lie between ~32768 and 32767 inclusive, width and height values must be between 0 and 65535 inclusive.

This means that `Rect {top,left,bottom,right}` must have `right >= left` and `bottom >= top` . Similarly, `Area {x,y,w,h}` must have `w >= 0` and `h >= 0` .

Where an **XRectangle** is used to pass width and height values only, the x and y members must both be 0.

2.6.2 XWindows

Types:

```
exception XWindows of string
```

Arguments:

"Display not connected"	Attempt to use X functions with no display connected.
"Non-existant resource"	Attempt to use a resource value from a previous session.

"Not a window"	Attempt to use a pixmap **Drawable** as a window
"Not a pixmap"	Attempt to use a window **Drawable** as a pixmap
"Handler mismatch"	Attempt to send a message to a window handler when the window has had a new handler installed with **XSetHandler**.
"<functionName> failed"	Xlib detected an error condition when executing <functionName>
"Bad<className> in <functionName>"	The X server detected an error condition occurred when executing <functionName>. This is only reported when running synchronously. For example, "BadMatch in XChangeWindowAttributes" .

Description:

exception **XWindows** is raised when an Xlib function returns an error condition.

2.7 Event Handling

2.7.1 IsCursorKey, IsFunctionKey, IsKeypadKey, IsMiscFunctionKey, IsModifierKey, IsPFKey

Types:

```
val IsCursorKey:       int -> bool
val IsFunctionKey:     int -> bool
val IsKeypadKey:       int -> bool
val IsMiscFunctionKey: int -> bool
val IsModifierKey:     int -> bool
val IsPFKey:           int -> bool
```

Description:

The **IsCursorKey** function returns true if the specified KeySym is a cursor key.

The **IsFunctionKey** function returns true if the KeySym is a function key.

The **IsKeypadKey** function returns true if the specified KeySym is a keypad key.

The **IsMiscFunctionKey** function returns true if the specified KeySym is a miscellaneous function key.

The **IsModifierKey** function returns true if the specified KeySym is a modifier key.

The **IsPFKey** function returns true if the specified KeySym is a PF key.

2.7.2 ShiftDown, ControlDown

Types:

```
val ShiftDown:   Modifier list -> bool
val ControlDown: Modifier list -> bool
```

Syntax:

```
ShiftDown   modifiers
ControlDown modifiers
```

Arguments:

modifiers	Specifies the modifiers from a key event

Description:

The **ShiftDown** convenience function returns true if **ShiftMask** is in the modifiers list, and false otherwise. This indicates if the Shift key was pressed when the key event was generated.

The **ControlDown** convenience function returns true if **ControlMask** is in the modifiers list, and false otherwise. This indicates if the Control key was pressed when the key event was generated.

2.7.3 XLookupString, NoSymbol

Types:

```
val XLookupString: int -> Modifier list -> (string * int)
val NoSymbol:      int
```

Syntax:

```
val (string,keysym) = XLookupString keycode modifiers ;
```

Arguments:

keycode	Specifies the keycode from a key event
modifiers	Specifies the modifiers from a key event
string	Returns the string for that combination
keysym	Returns the keysym for that combination

Description:

The **XLookupString** function translates a key event to a KeySym and a string. The KeySym is obtained by using the standard interpretation of the Shift, Lock, and group modifiers as defined in the X Protocol specification. If the KeySym has been rebound, the bound string will be returned. Otherwise, the KeySym is mapped, if possible, to an ISO Latin-1 character or (if the Control modifier is on) to an ASCII control character, and that character is returned. If no KeySym is defined for keycode, the KeySym returned is **NoSymbol**.

2.7.4 XSelectInput

Types:

```
val XSelectInput: Drawable -> EventMask list -> unit
```

Syntax:

```
XSelectInput w events ;
```

Arguments:

events	Specifies the list of events you wish to handle.
w	Specifies the window.

Argument Type:

```
datatype EventMask = KeyPressMask             | KeyReleaseMask
                   | ButtonPressMask          | ButtonReleaseMask
                   | EnterWindowMask          | LeaveWindowMask
                   | PointerMotionMask        | PointerMotionHintMask
                   | Button1MotionMask        | Button2MotionMask
                   | Button3MotionMask        | Button4MotionMask
                   | Button5MotionMask        | ButtonMotionMask
                   | KeymapStateMask          | ExposureMask
                   | VisibilityChangeMask     | StructureNotifyMask
                   | ResizeRedirectMask       | SubstructureNotifyMask
                   | SubstructureRedirectMask | FocusChangeMask
                   | PropertyChangeMask       | ColormapChangeMask
                   | OwnerGrabButtonMask      | ButtonClickMask
```

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 doNotPropagate attribute 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.

2.7.5 XSetHandler, NullHandler

Types:

```
val XSetHandler: Drawable ->
                 ('a XEvent * 'state -> 'state) -> 'state -> (int -> 'a -> unit)

val NullHandler: 'a XEvent * 'state -> 'state
```

Syntax:

```
val sender = XSetHandler w Handler initialState ;

sender delay message ;
```

Arguments:

w	Specifies the window.
Handler	Specifies the event handling function.
initialState	Specifies the initial state.
sender	Returns a function that can send a strongly typed message to the window at any specified time in the future.
delay	Specifies a delay in milliseconds before the message is sent.
message	Specifies the message value. The type of the message matches the type of the **XEvent** processed by the event handling function.

Description:

When a window is created it is initially unhandled. It can be used for drawing on, but it will not process any events. An ML function can then be registered for that window, and an initial value supplied. The registered function will transform the value to a new value every time an event arrives for that window. In other words, a functional state machine is set up for each window. We also implement strongly typed message passing between windows, and millisecond-resolution timer events.

XSetHandler installs a new event handling function for a window. Event handlers typically pattern-match on the **XEvent** members, choosing to match events that they are interested in, and then finish off with a default pattern match to provide a default action for all other events. For example:

```
fun Handler (Expose {window,region,...},state) = ...

|   Handler (EnterNotify {window,...},state) = ...
|   Handler (LeaveNotify {window,...},state) = ...

|   Handler (MotionNotify {window,pointer,...},state) = ...

|   Handler (_,state) = state ;  (* default is to do nothing *)
```

Underneath, we have a process that maintains a current state and an event handler for every window, and manages the events from the X server. As each event arrives it applies the handler for that window to the event and the current state, which returns a new state, which replaces the original state. Because only one process handles the events,

we guarantee that no other handler function can run at the same time. If the handler function raises an exception, instead of returning a new state, then the current state is left unchanged, and the exception is reported at the terminal. In this way all events are handled in turn in a predictable order, and in much the same way that other X toolkits work. The Poly/ML top level shell process is still available for debugging and control.

If a window has an operation that takes a long time to complete, then the programmer can use Poly/ML processes to do the computations 'in the background' and 'send' the result as a message to the window for display. However, the use of processes in this way is discouraged as they are not standard.

If a window function loops, then all other windows will freeze. Since the Poly/ML top level shell is available the user can type `^C` followed by 'f' to raise Interrupt in that window function.

The function returned by **XSetHandler** can be used to send messages to this window, if messages are not required then this function can be ignored. The message value will be wrapped up in a **Message XEvent** and passed to the event handling function, the type of the message value is guaranteed match the type of **XEvent** handled by the event handler. The time the message arrives can be modified using the delay parameter, which is the delay in milliseconds. This is often useful for implementing flashing displays, or auto-repeat functions.

2.7.6 XSetInputFocus, XGetInputFocus

Types:

```
val XSetInputFocus: Drawable -> RevertCode -> int -> unit
val XGetInputFocus: unit -> (Drawable * RevertCode)
```

Syntax:

```
XSetInputFocus focus revertTo time ;
val (focus,revertTo) = XGetInputFocus() ;
```

Arguments:

focus	Specifies or returns the window, **PointerRoot**, or **NoDrawable**.
revertTo	Specifies or returns 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**.

Argument Type:

```
datatype RevertCode = RevertToParent | RevertToPointerRoot | RevertToNone

val CurrentTime: int
```

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 **NoDrawable**, all keyboard events are discarded until a new focus window is set, and the revertTo 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 revertTo 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 revertTo argument to determine the new focus window as follows:

If revertTo is **RevertToParent**, the focus reverts to the parent (or the closest viewable ancestor), and the new revertTo value is taken to be **RevertToNone**.

If revertTo is **RevertToPointerRoot** or **RevertToNone**, the focus reverts to **PointerRoot** or **NoDrawable**, respectively. When the focus reverts, the X server generates **FocusIn** and **FocusOut** events, but the last-focus-change time is not affected.

The **XGetInputFocus** function returns the focus window and the current focus state.

2.7.7 XSync, XFlush

Types:

```
val XSync:  bool -> unit
val XFlush: unit -> unit
```

Syntax:

```
XSync discard ;
XFlush() ;
```

Arguments:

discard Specifies a bool that indicates whether **XSync** discards all events on the event queue.

Description:

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. Any events generated by the server are enqueued into the library's event queue.

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**.

The **XFlush** function flushes the output buffer. Most client applications need not use this function because the output buffer is automatically flushed internally as events are read.

2.7.8 XSyncronise, XSynchronize

Types:

```
val XSyncronise: int -> unit
```

Syntax:

```
XSyncronise flag ;
```

Arguments:

flag Specifies that synchronization is enabled or disabled

Description:

If flag is non-zero, XSynchronize turns on synchronous behavior. If flag is zero, XSynchronize turns off synchronous behavior.

NOTE that the current release has XSynchronize misspelled as **XSyncronise**.

2.7.9 XTranslateCoordinates

Types:

```
val XTranslateCoordinates: Drawable -> Drawable -> XPoint -> XPoint * Drawable
```

Syntax:

```
val (dstPoint,child) = XTranslateCoordinates srcWindow destWindow srcPoint ;
```

Arguments:

srcWindow	Specifies the source window.
destWindow	Specifies the destination window.
srcPoint	Specifies the x and y coordinates within the source window
dstPoint	Return the x and y coordinates within the destination window
child	Returns the child if the coordinates are contained in a mapped child of the destination window.

Description:

The **XTranslateCoordinates** function takes the srcPoint coordinates relative to the source window's origin and returns these coordinates to dstPoint relative to the destination window's origin. If **XTranslateCoordinates** returns zero, srcWindow and destWindow are on different screens, and dstPoint is (0,0). If the coordinates are contained in a mapped child of destWindow, that child is returned as child. Otherwise, child has the value **NoDrawable**.

2.8 Fonts

2.8.1 CharLBearing, CharRBearing, CharWidth, CharAscent, CharDescent, CharAttributes

Types:

```
val CharLBearing:   XCharStruct -> int
val CharRBearing:   XCharStruct -> int
val CharWidth:      XCharStruct -> int
val CharAscent:     XCharStruct -> int
val CharDescent:    XCharStruct -> int
val CharAttributes: XCharStruct -> int
```

Argument Type:

```
datatype XCharStruct = XCharStruct of { lbearing:   int,
                                        rbearing:   int,
                                        width:      int,
                                        ascent:     int,
                                        descent:    int,
                                        attributes: int }
```

Description:

These convenience functions return the individual fields of the **XCharStruct** datatype.

2.8.2 FSFont, FSDirection, FSMinChar, FSMaxChar, FSMinByte1, FSMaxByte1, FSAllCharsExist, FSAllCharsExist, FSDefaultChar, FSMinBounds, FSMaxBounds, PSPerChar, FSPerChar, FSAscent, FSDescent, FSAscent, FSDescent, FSMinWidth, FSMaxWidth, FSMinHeight, FSMaxHeight

Types:

```
val FSFont:          XFontStruct -> Font
val FSDirection:     XFontStruct -> FontDirection
val FSMinChar:       XFontStruct -> int
val FSMaxChar:       XFontStruct -> int
val FSMinByte1:      XFontStruct -> int
val FSMaxByte1:      XFontStruct -> int
```

```
val FSAllCharsExist: XFontStruct -> bool
val FSDefaultChar:   XFontStruct -> int
val FSMinBounds:     XFontStruct -> XCharStruct
val FSMaxBounds:     XFontStruct -> XCharStruct
val PSPerChar:       XFontStruct -> XCharStruct list
val FSAscent:        XFontStruct -> int
val FSDescent:       XFontStruct -> int

val FSMinWidth:  XFontStruct -> int
val FSMaxWidth:  XFontStruct -> int
val FSMinHeight: XFontStruct -> int
val FSMaxHeight: XFontStruct -> int
```

Argument Type:

```
datatype XFontStruct = XFontStruct of { font:          Font,
                                        direction:     FontDirection,
                                        minChar:       int,
                                        maxChar:       int,
                                        minByte1:      int,
                                        maxByte1:      int,
                                        allCharsExist: bool,
                                        defaultChar:   int,
                                        minBounds:     XCharStruct,
                                        maxBounds:     XCharStruct,
                                        perChar:       XCharStruct list,
                                        ascent:        int,
                                        descent:       int }
```

Description:

These convenience functions return the individual fields of the **XFontStruct** datatype. NOTE that the current release has FSPerChar misspelled as **PSPerChar**.
FSMinWidth, **FSMaxWidth**, **FSMinHeight** and **FSMaxHeight** are defined as:

```
fun FSMinWidth  f = CharWidth  (FSMinBounds f) ;
fun FSMaxWidth  f = CharWidth  (FSMaxBounds f) ;
fun FSMinHeight f = CharAscent (FSMinBounds f) + CharDescent (FSMinBounds f) ;
fun FSMaxHeight f = CharAscent (FSMaxBounds f) + CharDescent (FSMaxBounds f) ;
```

2.8.3 XListFonts, XListFontsWithInfo

Types:

```
val XListFonts:         string -> int -> string list
val XListFontsWithInfo: string -> int -> (string list * XFontStruct list)
```

Syntax:

```
val names = XListFonts pattern maxNames ;
val (names,fonts) = XListFontsWithInfo pattern maxNames ;
```

Arguments:

pattern	Specifies the pattern string that can contain wildcard characters.
maxNames	Specifies the maximum number of names to be returned.
names	Specifies the list of font names returned.
fonts	Specifies the list of font structures returned.

Description:

The **XListFonts** function returns a list 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. 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 list of names is limited to size specified by maxNames. If **XListFonts** fails then exception **XWindows** is raised with "XListFonts failed" .

The **XListFontsWithInfo** function returns a list of font names that match the specified pattern and their associated font information. The list of names is limited to size specified by maxNames. The information returned for each font is identical to what **XLoadQueryFont** would return except that the per-character metrics are not returned. The pattern string can contain any characters, but each asterisk "*" is a wildcard for any number of characters, and each question mark "?" is a wildcard for a single character. If **XListFontsWithInfo** fails then exception **XWindows** is raised with "XListFontsWithInfo failed" .

2.8.4 XLoadFont, XLoadQueryFont, XQueryFont, XFreeFont, XUnloadFont

Types:

```
val XLoadFont:      string -> Font
val XLoadQueryFont: string -> XFontStruct
val XQueryFont:     Font -> XFontStruct
val XFreeFont:      XFontStruct -> unit
val XUnloadFont:    Font -> unit
```

Syntax:

```
val font = XLoadFont name ;
val fs = XLoadQueryFont name ;
val fs = XQueryFont font ;
XFreeFont fs ;
XUnloadFont font ;
```

Arguments:

font	Specifies the font identifier.
fs	Specifies the font structure.
name	Specifies the name of the font.

Argument Type:

```
datatype FontDirection = FontLeftToRight | FontRightToLeft

datatype XCharStruct = XCharStruct of { lbearing:   int,
                                        rbearing:   int,
                                        width:      int,
                                        ascent:     int,
                                        descent:    int,
                                        attributes: int }

datatype XFontStruct = XFontStruct of { font:          Font,
                                        direction:     FontDirection,
                                        minChar:       int,
                                        maxChar:       int,
                                        minByte1:      int,
                                        maxByte1:      int,
                                        allCharsExist: bool,
                                        defaultChar:   int,
                                        minBounds:     XCharStruct,
                                        maxBounds:     XCharStruct,
                                        perChar:       XCharStruct list,
                                        ascent:        int,
                                        descent:       int }
```

Argument Description:

The **XFontStruct** structure contains all of the information for the font and consists of the font-specific information as well as a list of **XCharStruct** structures for the characters contained in the font.

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: byte 1 specifies the range of defined rows and byte 2 defines the range of defined columns of the font. Single byte/character fonts have one row defined, and the byte 2 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 defaultChar 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 minByte1 and maxByte1 members are both zero, minChar specifies the linear character index corresponding to the first element of the perChar list, and maxChar specifies the linear character index of the last element.

If either minByte1 or maxByte1 are non-zero, then both minChar and maxChar are less than 256, and the 2-byte character index values corresponding to the perChar list element N (counting from 0) are:

```
byte 1 = N div D + minByte1
byte 2 = N mod D + minChar
```

If the perChar list is empty, all glyphs between the first and last character indexes inclusive have the same information, as given by both minBounds and maxBounds.

If allCharsExist is true, all characters in the perChar list have non-zero bounding boxes.

The defaultChar member specifies the character that will be used when an undefined or nonexistent character is printed. The defaultChar is a 16-bit character (not a 2-byte character). For a font using 2-byte matrix format, the defaultChar has byte 1 in the most-significant byte and byte 2 in the least-significant byte. If the defaultChar itself specifies an undefined or nonexistent character, no printing is performed for an undefined or nonexistent character.

The minBounds and maxBounds members contain the most extreme values of each individual **XCharStruct** component over all elements of this list (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+minBounds.lbearing,y-maxBounds.ascent). Its width is (maxBounds.rbearing-minBounds.lbearing) and its height is (maxBounds.ascent+maxBounds.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) and 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.

Description:

The **XLoadFont** function loads the specified font and returns the **Font** value for it. The name should be ISO Latin-1 encoding; uppercase and lowercase do not matter. The interpretation of characters "?" and "*" in the name is not defined by the core protocol but is reserved for future definition. A structured format for font names is specified in the X Consortium standard X Logical **Font** Description Conventions. If the font does not exist then exception **XWindows** is raised with "XLoadFont failed" . 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**.

The **XQueryFont** function returns an **XFontStruct** structure, which contains information associated with the font. If **XQueryFont** fails then exception **XWindows** is raised with "XQueryFont failed" .

The **XLoadQueryFont** function provides the most common way for accessing a font. **XLoadQueryFont** both opens (loads) the specified font and returns the appropriate

XFontStruct structure. If the font does not exist then exception **XWindows** is raised with "XLoadQueryFont failed" .

The **XFreeFont** function deletes the association between the **Font** value in the **XFontStruct** and the specified font in the server. The font itself will be freed when no other resource references it. The **XFontStruct** and the font should not be referenced again.

The **XUnloadFont** function deletes the association between the **Font** value and the specified font in the server. The font itself will be freed when no other resource references it. The font should not be referenced again.

2.8.5 XSetFontPath, XGetFontPath

Types:

```
val XSetFontPath: string list -> unit
val XGetFontPath: unit -> string list
```

Syntax:

```
XSetFontPath directories ;
val directories = XGetFontPath() ;
```

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.

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 which are currently referenced by an application. The meaning of an error from this request is operating system dependent.

The **XGetFontPath** function returns a list of strings containing the search path. If **XGetFontPath** fails then exception **XWindows** is raised with "XGetFontPath failed" .

2.8.6 XTextExtents, XTextExtents16

Types:

```
val XTextExtents: XFontStruct ->
                  string      -> (FontDirection * int * int * XCharStruct)

val XTextExtents16: XFontStruct ->
                    int list    -> (FontDirection * int * int * XCharStruct)
```

Syntax:

```
val (direction,ascent,descent,overall) = XTextExtents fs string ;
val (direction,ascent,descent,overall) = XTextExtents16 fs bigChars ;
```

Arguments:

direction	Returns the value of the direction hint (**FontLeftToRight** or **FontRightToLeft**).
fs	Specifies the **XFontStruct** to use.
ascent	Returns the font ascent.
descent	Returns the font descent.
string	Specifies the character string.
bigChars	Specifies the character string as a list of 16 bit integers.
overall	Returns the overall size in a **XCharStruct** structure.

Description:

The **XTextExtents** and **XTextExtents16** functions perform the size computation locally using the **XFontStruct** provided. 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 2-byte matrix indexing rather than 16-bit linear indexing, the most-significant 8-bits of each int in bigChars is used as byte 1, and the least-significant 8-bits is used as byte 2.

If the font has no defined default character, undefined characters in the string are taken to have all zero metrics.

Characters with all zero metrics are ignored. If the font has no defined defaultChar, the undefined characters in the string are also ignored.

2.8.7 XTextWidth, XTextWidth16

Types:

```
val XTextWidth:   XFontStruct -> string   -> int
val XTextWidth16: XFontStruct -> int list -> int
```

Syntax:

```
val width = XTextWidth fs string ;
val width = XTextWidth16 fs bigChars ;
```

Arguments:

fs	Specifies the **XFontStruct** to use.
string	Specifies the character string.
bigChars	Specifies the character string as a list of 16 bit integers.
width	Returns the width in pixels.

Description:

The **XTextWidth** and **XTextWidth16** functions return the width of the specified 8-bit or 2-byte character strings.

2.9 Geometry

2.9.1 AddPoint, SubtractPoint

Types:

```
infix AddPoint SubtractPoint

val AddPoint:      (XPoint * XPoint) -> XPoint
val SubtractPoint: (XPoint * XPoint) -> XPoint
```

Description:

AddPoint takes two points and adds the x coordinates together and the y coordinates together to make the resulting point. In vector arithmetic this is equivalent to adding two vectors.

SubtractPoint subtracts the x coordinate of the second point from the x coordinate of the first, and subtracts the y coordinate of the second point from the y coordinate of the first. In vector arithmetic this is equivalent to vector subtraction.

2.9.2 Inside, Overlap, Within, LeftOf, RightOf, AboveOf, BelowOf, HorizontallyAbutting, VerticallyAbutting

Types:

```
infix Inside Overlap Within
infix LeftOf RightOf AboveOf BelowOf
infix HorizontallyAbutting VerticallyAbutting

val Inside:               (XRectangle * XRectangle) -> bool
val Overlap:              (XRectangle * XRectangle) -> bool
val Within:               (XPoint     * XRectangle) -> bool
val LeftOf:               (XPoint     * XRectangle) -> bool
val RightOf:              (XPoint     * XRectangle) -> bool
val AboveOf:              (XPoint     * XRectangle) -> bool
val BelowOf:              (XPoint     * XRectangle) -> bool
val HorizontallyAbutting: (XRectangle * XRectangle) -> bool
val VerticallyAbutting:   (XRectangle * XRectangle) -> bool
```

Description:

`a Inside b` is true if rectangle a is totally enclosed by b.

`a Overlap b` is true if the two rectangles intersect.

`a Within b` is true if point a is inside rectangle b.

`a LeftOf b` is true if point a is outside and to the left of rectangle b.

`a RightOf b` is true if point a is outside and to the right of rectangle b.

`a AboveOf b` is true if point a is outside and above rectangle b.

`a BelowOf b` is true if point a is outside and below rectangle b.

`a HorizontallyAbutting b` is true if the left edge of a touches the right edge of b, or the right edge of a touches the left edge of b.

`a VerticallyAbutting b` is true if the top edge of a touches the bottom edge of b, or the bottom edge of a touches the top edge of b.

2.9.3 Intersection, Union, Section

Types:

```
val Intersection: XRectangle -> XRectangle  -> Section
val Union:        XRectangle -> XRectangle  -> XRectangle
```

Argument Type:

```
datatype Section = Nothing | Section of XRectangle
```

Description:

Intersection computes the intersection of the two rectangles. If the rectangles do not intersect then it returns **Nothing**, otherwise it returns **Section** of the intersection.

Union computes the bounding rectangle for the union of the two rectangles.

2.9.4 Left, Right, Top, Bottom, Width, Height, TopLeft, TopRight, BottomLeft, BottomRight, XRectangle, Area, Rect, DestructRect, DestructArea, empty

Types:

```
eqtype XRectangle

val Rect: {left:int,right:int,top:int,bottom:int} -> XRectangle
val Area: {x:int,y:int,w:int,h:int}               -> XRectangle

val DestructRect: XRectangle -> {left:int,right:int,top:int,bottom:int}
val DestructArea: XRectangle -> {x:int,y:int,w:int,h:int}

exception XRectangle of {top:int,left:int,bottom:int,right:int}

val Left:   XRectangle -> int
```

```
val Right:  XRectangle -> int
val Top:    XRectangle -> int
val Bottom: XRectangle -> int
val Width:  XRectangle -> int
val Height: XRectangle -> int

val TopLeft:     XRectangle -> XPoint
val TopRight:    XRectangle -> XPoint
val BottomLeft:  XRectangle -> XPoint
val BottomRight: XRectangle -> XPoint

val empty = Area {x=0,y=0,w=0,h=0}
```

Syntax:

```
val area = Area { x = 0, y = 0, w = 100, h = 200 } ;
val {x,y,w,h} = DestructArea area ;
val left = Left area ;
```

Description:

XRectangles are used to represent pixel areas. For example, an **Expose** event on a window will contain the position and size of the rectangular area which needs refreshing. XRectangles may also represent size only. For example, the coordinate system of a window is represented as an **XRectangle** which has width and height, but the top left corner of the rectangle is at (0,0).

XRectangles representing pixel areas can be thought of in two ways.

The first way is to call the top left pixel in the rectangle `(x,y)` and the width and height of the rectangle are `(width,height)` . Then, an empty rectangle has `width = 0` and `height = 0` , and the point `(a,b)` is in a non-empty rectangle only if `a >= x` and `a < (x+width)` and `b >= y` and `b < (y+height)` .

The second way is to call the top left pixel inside the area `(top,left)` and to call the outside bottom right pixel `(bottom,right)` . Then, the empty rectangle has `(top,left) = (bottom,right)` , and the point `(x,y)` is in a non-empty rectangle if `x >= left` and `x < right` and `y >= top` and `y < bottom` . `NOTE` that in X, y coordinates increase `down` the screen, so `top <= bottom` .

You should be careful not to generate coordinates out of range. x and y coordinates must lie between ˜32768 and 32767 inclusive, width and height values must be between 0 and 65535 inclusive.

This means that `Rect {top,left,bottom,right}` must have `right >= left` and `bottom >= top` . Similarly, `Area {x,y,w,h}` must have `w >= 0` and `h >= 0` . If these constraints are not met then exception **XRectangle** is raised.

Convenience functions exist to destruct XRectangles. **Left**, **Right**, **Top** and **Bottom** return the single coordinate for an edge of an **XRectangle**. **Width** and **Height** return the width and height of an **XRectangle**. **TopLeft**, **TopRight**, **BottomLeft** and **BottomRight** return the coordinates of a corner of an **XRectangle** as points.

`empty` is a rectangle with zero area.

2.9.5 MakeRect, SplitRect

Types:

```
val MakeRect:  XPoint -> XPoint -> XRectangle
val SplitRect: XRectangle  -> (XPoint * XPoint)
```

Syntax:

```
val (topLeft,bottomRight) = SplitRect r ;
val r = MakeRect corner1 corner2 ;
```

Description:

MakeRect constructs an **XRectangle** given two points corresponding to any pair of opposite corners of the rectangle. This is useful when the order of the two points is not known, for example when dragging a rubber-banded box on the screen.

SplitRect returns the pair of points corresponding to the top-left and bottom-right corners of the **XRectangle**. It will always be the case that `left <= right` and `top <= bottom` .

2.9.6 NegativePoint

Types:

```
val NegativePoint: XPoint -> XPoint
```

Description:

NegativePoint negates both the x and y coordinates of the point. This is equivalent to reflecting about the x axis and the y axis.

2.9.7 OutsetRect, OffsetRect, IncludePoint

Types:

```
val OutsetRect:   int -> XRectangle -> XRectangle
val OffsetRect:   XRectangle -> XPoint -> XRectangle
val IncludePoint: XPoint -> XRectangle -> XRectangle
```

Description:

`OutsetRect n R` takes rectangle R and expands its area by n units in all four directions. Typically n is positive and this function is used to expand areas to incorporate borders of the same width all around. With a negative n it can be used to shrink an area towards the centre of the area. If n is more negative than half the width or height of the area then exception **XRectangle** is raised.

`OffsetRect R (XPoint{x,y})` adds x to both x coordinates and y to both y coordinates of R. This is typically used to move a rectangle by an (x,y) offset or vector.

`IncludePoint p R` is used to expand the area of R to include the point p. If p is already inside the rectange R then R is returned unchanged. If p is outside the rectangle R then R is expanded in the direction of p so that p is now just inside R.

2.9.8 Reflect

Types:

```
val Reflect: XRectangle -> XRectangle
```

Description:

Reflect takes an **XRectangle** and swaps the x and y coordinates over; left is swapped with top and right is swapped with bottom. This is equivalent to reflecting the points about the 45-degree line that has the equation y = x.

2.9.9 XPoint

Types:

```
datatype XPoint = XPoint of { x:int,y:int }

val origin = XPoint {x=0,y=0}
```

Syntax:

```
XPoint { x=100,y=200 }
```

Description:

XPoints are used to represent the coordinates of pixels. For example, the position of the top left pixel of a window on the screen is represented as an **XPoint**.

You should be careful not to generate coordinates out of range. x and y coordinates must lie between ~32768 and 32767 inclusive.

`origin` is the point (0,0), and is typically used to refer to the origin of the coordinate system. In X, the origin is the top left corner of a window.

2.10 GC - Graphics Context

2.10.1 DefaultGC

Types:

```
val DefaultGC: unit -> GC
```

Syntax:

```
val gc = DefaultGC() ;
```

Description:

The **DefaultGC** function returns the default **GC** for the root window of the screen.

2.10.2 XCreateGC, XChangeGC, XFreeGC

Types:

```
val XCreateGC: Drawable -> XGCValue list -> GC
val XChangeGC: GC -> XGCValue list -> unit
val XFreeGC:   GC -> unit
```

Syntax:

```
val gc = XCreateGC d values ;
XChangeGC gc values ;
XFreeGC gc ;
```

Arguments:

d	Specifies the drawable.
gc	Specifies the **GC**.
values	Specifies which components in the **GC** are to be set or changed.

Argument Type:

```
datatype XGCValue = GCFunction          of GCFunction
                  | GCPlaneMask         of int
                  | GCForeground        of int
                  | GCBackground        of int
                  | GCLineWidth         of int
                  | GCLineStyle         of GCLineStyle
                  | GCCapStyle          of GCCapStyle
                  | GCJoinStyle         of GCJoinStyle
                  | GCFillStyle         of GCFillStyle
                  | GCFillRule          of GCFillRule
                  | GCTile              of Drawable
                  | GCStipple           of Drawable
                  | GCTSOrigin          of XPoint
                  | GCFont              of Font
                  | GCSubwindowMode     of GCSubwindowMode
                  | GCGraphicsExposures of bool
                  | GCClipOrigin        of XPoint
                  | GCClipMask          of Drawable
                  | GCDashOffset        of int
                  | GCDashList          of int
                  | GCArcMode           of GCArcMode
```

Argument Description:

The **GCFunction** 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 colour display, but special applications may use other functions, particularly in concert with particular planes of a colour display. The 16 **GC** functions are:

```
datatype GCFunction = GXclear        | GXand        | GXandReverse | GXcopy
                    | GXandInverted  | GXnoop       | GXxor        | GXor
                    | GXnor          | GXequiv      | GXinvert     | GXorReverse
                    | GXcopyInverted | GXorInverted | GXnand       | GXset
```

Many graphics operations depend on either pixel values or planes in a **GC**. The **GC-PlaneMask** attribute is an int, 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. The value **AllPlanes** can be used to refer to all planes of the screen simultaneously. The result is computed by the following:

```
((src GC-FUNCTION 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.

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.

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.

```
datatype GCLineStyle = LineSolid | LineOnOffDash | LineDoubleDash
```

The line-style defines which sections of a line are drawn:

LineSolid The full path of the line is drawn.

LineDoubleDash The full path of the line is drawn, but the even dashes are filled differently than the odd dashes (see fill-style) with **CapButt** style used where even and odd dashes meet.

LineOnOffDash Only the even dashes are drawn, and cap-style applies to all internal ends of the individual dashes, except **CapNotLast** is treated as **CapButt**.

```
datatype GCCapStyle = CapNotLast | CapButt | CapRound | CapProjecting
```

The cap-style defines how the endpoints of a path are drawn:

CapNotLast This is equivalent to **CapButt** except that for a line-width of zero the final endpoint is not drawn.

CapButt The line is square at the endpoint (perpendicular to the slope of the line) with no projection beyond.

CapRound The line has a circular arc with the diameter equal to the line-width, centered on the endpoint. (This is equivalent to **CapButt** for line-width of zero).

CapProjecting The line is square at the end, but the path continues beyond the endpoint for a distance equal to half the line-width. (This is equivalent to **CapButt** for line-width of zero).

```
datatype GCJoinStyle = JoinMiter | JoinRound | JoinBevel
```

The join-style defines how corners are drawn for wide lines:

JoinMiter The outer edges of two lines extend to meet at an angle. However, if the angle is less than 11 degrees, then a **JoinBevel** join-style is used instead.

JoinRound The corner is a circular arc with the diameter equal to the line-width, centered on the joinpoint.

JoinBevel The corner has **CapButt** endpoint styles with the triangular notch filled.

For a line with coincident endpoints (x1=x2,y1=y2), when the cap-style is applied to both endpoints, the semantics depends on the line-width and the cap-style:

CapNotLast	**thin**	The results are device-dependent, but the desired effect is that nothing is drawn.
CapButt	**thin**	The results are device-dependent, but the desired effect is that a single pixel is drawn.
CapRound	**thin**	The results are the same as for **CapButt**/thin.
CapProjecting	**thin**	The results are the same as for **CapButt**/thin.
CapButt	**wide**	nothing is drawn.
CapRound	**wide**	The closed path is a circle, centered at the endpoint, and with the diameter equal to the line-width.
CapProjecting	**wide**	The closed path is a square, aligned with the coordinate axes, centered at the endpoint, and with the sides equal to the line-width.

For a line with coincident endpoints (x1=x2, y1=y2), when the join-style is applied at one or both endpoints, the effect is as if the line was removed from the overall path. However, if the total path consists of or is reduced to a single point joined with itself, the effect is the same as when the cap-style is applied at both endpoints.

The tile/stipple 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.

```
datatype GCFillStyle = FillSolid    | FillTiled
                     | FillStippled | FillOpaqueStippled
```

The fill-style defines the contents of the source for line, text, and fill requests. For all text and fill requests, for line requests with line-style **LineSolid**, and for the even dashes for line requests with line-style **LineOnOffDash** or **LineDoubleDash**, the following apply:

FillSolid	Foreground
FillTiled	Tile
FillOpaqueStippled	A tile with the same width and height as stipple, but with background everywhere stipple has a zero and with foreground everywhere stipple has a one
FillStippled	Foreground masked by stipple

When drawing lines with line-style **LineDoubleDash**, the odd dashes are controlled by the fill-style in the following manner:

FillSolid	Background
FillTiled	Same as for even dashes
FillOpaqueStippled	Same as for even dashes
FillStippled	Background masked by stipple

Storing a pixmap in a **GC** might or might not result in a copy being made. If the pixmap is later used as the destination for a graphics request, the change might or might not be reflected in the **GC**. If the pixmap is used simultaneously in a graphics request both as a destination and as a tile or stipple, the results are undefined.

For optimum performance, you should draw as much as possible with the same **GC** (without changing its components). The costs of changing **GC** components relative to using

different GCs depend upon the display hardware and the server implementation. It is quite likely that some amount of **GC** information will be cached in display hardware and that such hardware can only cache a small number of GCs.

The dashes value is actually a simplified form of the more general patterns that can be set with **XSetDashes**. Specifying a value of N is equivalent to specifying the two-element list [N,N] in **XSetDashes**. The value must be non-zero, or a **BadValue** error results. The value must be less than 256 or exception Range is raised.

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 **NoDrawable**, 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.

```
datatype GCSubwindowMode = ClipByChildren | IncludeInferiors
```

You can set the subwindow-mode to **ClipByChildren** or **IncludeInferiors**. For **ClipByChildren**, both source and destination windows are additionally clipped by all viewable **InputOutputClass** 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.

```
datatype GCFillRule = EvenOddRule | WindingRule
```

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).

```
datatype GCArcMode = ArcChord | ArcPieSlice
```

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).

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.

The **XChangeGC** function changes the components specified by values 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.

The **XFreeGC** function destroys the specified **GC**.

2.10.3 XSetArcMode

Types:

```
val XSetArcMode: GC -> GCArcMode -> unit
```

Syntax:

```
XSetArcMode gc mode ;
```

Arguments:

gc	Specifies the **GC**.
mode	Specifies the arc mode. You can pass **ArcChord** or **ArcPieSlice**.

Argument Type:

```
datatype GCArcMode = ArcChord | ArcPieSlice
```

Description:

The **XSetArcMode** function sets the arc mode in the specified **GC**.

2.10.4 XSetBackground

Types:

```
val XSetBackground: GC -> int -> unit
```

Syntax:

```
XSetBackground gc background ;
```

Arguments:

background	Specifies the background pixel.
gc	Specifies the **GC**.

Description:

The **XSetBackground** function sets the background pixel in the specified **GC**.

2.10.5 XSetClipMask

Types:

```
val XSetClipMask: GC -> Drawable -> unit
```

Syntax:

```
XSetClipMask gc pixmap ;
```

Arguments:

gc	Specifies the **GC**.
pixmap	Specifies the pixmap or **NoDrawable**.

Description:

The **XSetClipMask** function sets the clip-mask in the specified **GC** to the specified pixmap. If the clip-mask is set to **NoDrawable**, the pixels are are always drawn (regardless of the clip-origin).

2.10.6 XSetClipOrigin

Types:

```
val XSetClipOrigin: GC -> XPoint -> unit
```

Syntax:

```
XSetClipOrigin gc origin ;
```

Arguments:

gc	Specifies the **GC**.
origin	Specifies the x and y coordinates of the clip-mask origin.

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.

2.10.7 XSetClipRectangles

Types:

```
val XSetClipRectangles: GC -> XPoint -> XRectangle list -> GCOrder -> unit
```

Syntax:

```
XSetClipRectangles gc origin rectangles ordering ;
```

Arguments:

gc	Specifies the **GC**.
origin	Specifies the x and y coordinates of the clip-mask origin.
rectangles	Specifies a list of rectangles that define the clip-mask.
ordering	Specifies the ordering relations on the rectangles. You can pass **Unsorted**, **YSorted**, **YXSorted**, or **YXBanded**.

Argument Type:

```
datatype GCOrder = Unsorted | YSorted | YXSorted | YXBanded
```

Description:

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 **NoDrawable** 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.

2.10.8 XSetColours

Types:

```
val XSetColours: GC -> int -> int -> unit
```

Syntax:

```
XSetColours gc foreground background ;
```

Arguments:

background	Specifies the background pixel.
foreground	Specifies the foreground pixel.
gc	Specifies the **GC**.

Description:

The **XSetColours** convenience function sets the foreground and background components for the specified **GC**.

2.10.9 XSetDashes

Types:

```
val XSetDashes: GC -> int -> int list -> unit
```

Syntax:

```
XSetDashes offset dashes ;
```

Arguments:

dashes	Specifies the dash-list for the dashed line-style you want to set for the specified **GC**.
offset	Specifies the phase of the pattern for the dashed line-style you want to set for the specified **GC**.

Description:

The **XSetDashes** function sets the dash-offset and dash-list attributes for dashed line styles in the specified **GC**. There must be at least one element in the specified dash-list, or a **BadValue** error results. The initial and alternating elements (second, fourth, and so on) of the dash-list are the even dashes, and the others are the odd dashes. Each element specifies a dash length in pixels. All of the elements must be non-zero, or a **BadValue** error results. All of the elements must be less than 256 or exception Range is raised. Specifying an odd-length list is equivalent to specifying the same list concatenated with itself to produce an even-length list.

The dash-offset defines the phase of the pattern, specifying how many pixels into the dash-list the pattern should actually begin in any single graphics request. Dashing is continuous through path elements combined with a join-style but is reset to the dash-offset between each sequence of joined lines.

The unit of measure for dashes is the same for the ordinary coordinate system. Ideally, a dash length is measured along the slope of the line, but implementations are only required to match this ideal for horizontal and vertical lines. Failing the ideal semantics, it is suggested that the length be measured along the major axis of the line. The major axis is defined as the x axis for lines drawn at an angle of between -45 and +45 degrees or between 315 and 225 degrees from the x axis. For all other lines, the major axis is the y axis.

2.10.10 XSetFillRule

Types:

```
val XSetFillRule: GC -> GCFillRule -> unit
```

Syntax:

```
XSetFillRule gc rule ;
```

Arguments:

gc Specifies the **GC**.

rule Specifies the fill-rule you want to set for the specified **GC**. You can pass **EvenOddRule** or **WindingRule**.

Argument Type:

```
datatype GCFillRule = EvenOddRule | WindingRule
```

Description:

The **XSetFillRule** function sets the fill-rule in the specified **GC**.

2.10.11 XSetFillStyle

Types:

```
val XSetFillStyle: GC -> GCFillStyle -> unit
```

Syntax:

```
XSetFillStyle gc style ;
```

Arguments:

gc Specifies the **GC**.

style Specifies the fill-style you want to set for the specified **GC**. You can pass **FillSolid**, **FillTiled**, **FillStippled**, or **FillOpaqueStippled**.

Argument Type:

```
datatype GCFillStyle = FillSolid    | FillTiled
                     | FillStippled | FillOpaqueStippled
```

Description:

The **XSetFillStyle** function sets the fill-style in the specified **GC**.

2.10.12 XSetFont

Types:

```
val XSetFont: GC -> Font -> unit
```

Syntax:

```
XSetFont gc font ;
```

Arguments:

gc Specifies the **GC**.

font Specifies the font.

Description:

The **XSetFont** function sets the current font in the specified **GC**.

2.10.13 XSetForeground

Types:

```
val XSetForeground: GC -> int -> unit
```

Syntax:

```
XSetForeground gc foreground ;
```

Arguments:

foreground	Specifies the foreground pixel.
gc	Specifies the **GC**.

Description:

The **XSetForeground** function sets the foreground pixel in the specified **GC**.

2.10.14 XSetFunction

Types:

```
val XSetFunction: GC -> GCFunction  -> unit
```

Syntax:

```
XSetFunction gc function ;
```

Arguments:

function	Specifies the drawing function.
gc	Specifies the **GC**.

Description:

XSetFunction sets the drawing function in the specified **GC**.

2.10.15 XSetGraphicsExposures

Types:

```
val XSetGraphicsExposures: GC -> bool -> unit
```

Syntax:

```
XSetGraphicsExposures gc exposures ;
```

Arguments:

gc	Specifies the **GC**.
exposures	Specifies a bool that indicates whether you want **GraphicsExpose** and **NoExpose** events to be reported when calling **XCopyArea** and **XCopyPlane** with this **GC**.

Description:

The **XSetGraphicsExposures** function sets the graphics-exposures flag in the specified **GC**.

2.10.16 XSetLineAttributes

Types:

```
val XSetLineAttributes: GC -> int         ->
                           GCLineStyle ->
                           GCCapStyle  ->
                           GCJoinStyle -> unit
```

Syntax:

```
XSetLineAttributes gc lineWidth lineStyle capStyle joinStyle ;
```

Arguments:

capStyle	Specifies the line-style and cap-style you want to set for the specified **GC**. You can pass **CapNotLast**, **CapButt**, **CapRound**, or **CapProjecting**.
joinStyle	Specifies the line join-style you want to set for the specified **GC**. You can pass **JoinMiter**, **JoinRound**, or **JoinBevel**.
lineStyle	Specifies the line-style you want to set for the specified **GC**. You can pass **LineSolid**, **LineOnOffDash**, or **LineDoubleDash**.
lineWidth	Specifies the line-width you want to set for the specified **GC**.

Description:

The **XSetLineAttributes** function sets the line drawing components in the specified **GC**.

2.10.17 XSetPlaneMask

Types:

```
val XSetPlaneMask: GC -> int -> unit
```

Syntax:

```
XSetPlaneMask gc planeMask ;
```

Arguments:

gc	Specifies the **GC**.
planeMask	Specifies the plane mask.

Description:

The **XSetPlaneMask** function sets the plane mask in the specified **GC**.

2.10.18 XSetState

Types:

```
val XSetState: GC -> int -> int -> GCFunction -> int -> unit
```

Syntax:

```
XSetState gc foreground background function planeMask ;
```

Arguments:

background	Specifies the background pixel.
foreground	Specifies the foreground pixel.
function	Specifies the drawing function.
gc	Specifies the **GC**.
planeMask	Specifies the plane mask.

Description:

The **XSetState** function sets the foreground, background, plane mask, and function components for the specified **GC**.

2.10.19 XSetStipple

Types:

```
val XSetStipple: GC -> Drawable -> unit
```

Syntax:

```
XSetStipple gc stipple ;
```

Arguments:

gc	Specifies the **GC**.
stipple	Specifies the stipple you want to set for the specified **GC**.

Description:

The **XSetStipple** function sets the stipple in the specified **GC**. The stipple and **GC** must have the same depth, or a **BadMatch** error results.

2.10.20 XSetSubwindowMode

Types:

```
val XSetSubwindowMode: GC -> GCSubwindowMode -> unit
```

Syntax:

```
XSetSubwindowMode gc mode ;
```

Arguments:

gc	Specifies the **GC**.
mode	Specifies the subwindow mode. You can pass **ClipByChildren** or **IncludeInferiors**.

Argument Type:

```
datatype GCSubwindowMode = ClipByChildren | IncludeInferiors
```

Description:

The **XSetSubwindowMode** function sets the subwindow mode in the specified **GC**.

2.10.21 XSetTile

Types:

```
val XSetTile: GC -> Drawable -> unit
```

Syntax:

```
XSetTile gc tile ;
```

Arguments:

gc	Specifies the **GC**.
tile	Specifies the fill tile you want to set for the specified **GC**.

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.

2.10.22 XSetTSOrigin

Types:

```
val XSetTSOrigin: GC -> XPoint -> unit
```

Syntax:

```
XSetTSOrigin gc origin ;
```

Arguments:

gc	Specifies the **GC**.
origin	Specifies the x and y coordinates of the tile and stipple origin.

Description:

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.

2.11 Images

2.11.1 ImageByteOrder, ImageDepth, ImageSize

Types:

```
val ImageByteOrder: XImage -> ImageOrder
val ImageDepth:     XImage -> int
val ImageSize:      XImage -> XRectangle
```

Argument Type:

```
datatype ImageOrder = LSBFirst | MSBFirst
```

Syntax:

```
val order = ImageByteOrder image ;
val depth = ImageDepth image ;
val area  = ImageSize image ;
```

Description:

The **ImageByteOrder** function returns the byte order value of an **XImage**.

The **ImageSize** function returns the size in pixels of an **XImage**.

The **ImageDepth** function returns the depth value of an **XImage**.

2.11.2 VisualRedMask, VisualGreenMask, VisualBlueMask

Types:

```
val VisualRedMask:   Visual -> int
val VisualGreenMask: Visual -> int
val VisualBlueMask:  Visual -> int
```

Syntax:

```
val redMask   = VisualRedMask   visual ;
val greenMask = VisualGreenMask visual ;
val blueMask  = VisualBlueMask  visual ;
```

Arguments:

visual Specifies the visual.

Description:

These functions return the masks used for Z format images.

2.11.3 XCreateImage, XGetPixel, XPutPixel, XSubImage, XAddPixel

Types:

```
val XGetPixel: XImage -> XPoint -> int
val XPutPixel: XImage -> XPoint -> int -> unit
val XSubImage: XImage -> XRectangle -> XImage
val XAddPixel: XImage -> int -> unit

val XCreateImage: Visual      -> int ->
                  ImageFormat -> int ->
                  string      -> XRectangle -> int -> int -> XImage
```

Syntax:

```
val image = XCreateImage visual depth format offset
                         data area bitmapPad bytesPerLine ;

val pixel = XGetPixel image point ;
val image = XSubImage image subArea ;

XPutPixel image point pixel ;
XAddPixel image value ;
```

Arguments:

bitmapPad Specifies the quantum of a scanline (8, 16, or 32 bits). 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.

bytesPerLine	Specifies the number of bytes in the client image between the start of one scanline and the start of the next.
data	Specifies the image data.
depth	Specifies the depth of the image.
format	Specifies the format for the image. You can pass **XYBitmap**, **XYPixmap**, or **ZPixmap**.
area	Specifies the width and 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.
subArea	Specifies the position and size of the new subimage, in pixels.
value	Specifies the constant value that is to be added.
visual	Specifies the visual.
ximage	Specifies the image.
point	Specifies the x and y coordinates.

Argument Type:

```
datatype ImageFormat = XYBitmap | XYPixmap | ZPixmap

datatype ImageOrder = LSBFirst | MSBFirst

type ImageData

val Data: string -> ImageData

datatype XImage = XImage of { data:            ImageData,
                              size:            XRectangle,
                              depth:           int,
                              format:          ImageFormat,
                              xoffset:         int,
                              bitmapPad:       int,
                              byteOrder:       ImageOrder,
                              bitmapUnit:      int,
                              bitsPerPixel:    int,
                              bytesPerLine:    int,
                              visualRedMask:   int,
                              bitmapBitOrder:  ImageOrder,
                              visualBlueMask:  int,
                              visualGreenMask: int }
```

Description:

The **XCreateImage** function initializes the **XImage** byteOrder, bitmapBitOrder, and bitmapUnit values from the display and returns an **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 bytesPerLine, Xlib assumes that the scanlines are contiguous in memory and calculates the value of bytesPerLine itself.

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 int 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 int 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. The data is copied from the source image, and the image must contain the rectangle defined by subArea. If **XSubImage** fails then exception **XWindows** is raised with "XSubImage failed" .

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 colour resources and need to manipulate the image to that form.

2.11.4 XPutImage, XGetImage, XGetSubImage

Types:

```
val XPutImage: Drawable -> GC -> XImage -> XPoint -> XRectangle -> unit
val XGetImage: Drawable -> XRectangle -> int -> ImageFormat -> XImage

val XGetSubImage: Drawable -> XRectangle -> int -> ImageFormat ->
                  XImage -> XPoint -> unit
```

Syntax:

```
val image = XGetImage d area planeMask format ;
XGetSubImage d area planeMask format destImage destPoint ;
XPutImage d gc image srcPoint destArea ;
```

Arguments:

d	Specifies the drawable.
destImage	Specifies the destination image.
destPoint	Specifies 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.
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.
planeMask	Specifies the plane mask.
srcPoint	Specifies the offsets from the left and top edges of the image defined by the **XImage** structure.
area	Specifies the position and size of the subimage,

destArea Specifies coordinates relative to the origin of the drawable to define the destination 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 srcPoint and area arguments is drawn on the specified part of the drawable at the position specified by destArea

This function uses these **GC** components: foreground, background, function, plane-mask, subwindow-mode, clip-origin, and clip-mask.

The **XGetImage** function returns 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 planeMask argument. If the planeMask 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 planeMask argument. The function performs no range checking on the values in planeMask 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 planeMask.

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. If **XGetImage** fails then exception **XWindows** is raised with "XGetImage failed" .

The **XGetSubImage** function updates destImage 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 planeMask argument. If the format argument is **ZPixmap**, **XGetSubImage** returns as zero the bits in all planes not specified in the planeMask argument. The function performs no range checking on the values in planeMask and ignores extraneous bits.

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.

2.12 Properties and Selections

2.12.1 XDeleteProperty

Types:

```
val XDeleteProperty: Drawable -> int -> unit
```

Syntax:

```
XDeleteProperty w property ;
```

Arguments:

property	Specifies the property name.
w	Specifies the window containing the property.

Description:

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.

2.12.2 XInternAtom, XGetAtomName

Types:

```
val XInternAtom: string -> bool -> int
val XGetAtomName: int -> string
```

Syntax:

```
val atom = XInternAtom name onlyIfExists ;
val name = XGetAtomName atom ;
```

Arguments:

atom	Specifies the atom whose name you want returned.
name	Specifies the name associated with the atom you want returned.
onlyIfExists	Specifies a bool that indicates whether **XInternAtom** creates the atom.

Description:

The **XInternAtom** function returns the atom identifier associated with the specified name string. If onlyIfExists is false, the atom is created if it does not exist, otherwise, **XInternAtom** returns zero. You should use an ISO Latin-1 string for 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.

The **XGetAtomName** function returns the name associated with the specified atom. If **XGetAtomName** fails then exception **XWindows** is raised with "XGetAtomName failed" .

2.12.3 XSetProperty, XGetTextProperty

Types:

```
val XSetProperty: Drawable -> int -> PropertyValue -> int -> unit
val XGetTextProperty: Drawable -> int -> (string * int * int * int)
```

Syntax:

```
XSetProperty w propertyAtom propertyValue propertyTypeAtom ;
val (value,encoding,format,nitems) = XGetTextProperty w propertyAtom ;
```

Arguments:

w	Specifies the window
propertyAtom	Specifies the property name as an Atom.
propertyValue	Specifies the property value as one of the predefined types.
propertyTypeAtom	Specifies the name of the property type as an Atom.
value	Returns the contents of the property as chars/bytes.
encoding	Returns the property type atom
format	Returns the property format which is 1, 2 or 4 bytes per item.
nitems	Returns the number of items in the value

Argument Type:

```
datatype PropertyValue = PropertyArc         of XArc list
                       | PropertyAtom        of int list
                       | PropertyBitmap      of Drawable list
                       | PropertyColormap    of Colormap list
                       | PropertyCursor      of Cursor list
                       | PropertyDrawable    of Drawable list
                       | PropertyFont        of Font list
                       | PropertyInteger     of int list
                       | PropertyPixmap      of Drawable list
                       | PropertyPoint       of XPoint list
                       | PropertyRectangle   of XRectangle list
                       | PropertyRGBColormap of XStandardColormap list
                       | PropertyString      of string
```

```
| PropertyVisual      of Visual list
| PropertyWindow      of Drawable list
| PropertyWMHints     of XWMHint list
| PropertyWMSizeHints of XWMSizeHint list
| PropertyWMIconSizes of (XRectangle *
                          XRectangle *
                          XRectangle) list
```

Properties:

WM_CLIENT_MACHINE	The string name of the machine on which the client application is running.
WM_COMMAND	The command and arguments, separated by ASCII nulls, used to invoke the application.
WM_ICON_NAME	Name to be used in icon.
WM_NAME	Name of the application.

Description:

The **XSetProperty** function replaces the existing, specified property for the named window with the value and type specified. If the property does not already exist, **XSetProperty** creates it for the specified window.

The **XGetTextProperty** function reads the specified property from the window. The particular interpretation of the property's encoding and value as 'text' is left to the calling application. If the specified property does not exist on the window, then exception **XWindows** is raised with "XGetTextProperty failed" .

2.12.4 XSetSelectionOwner, XGetSelectionOwner, XConvertSelection, XSendSelectionNotify

Types:

```
val XSetSelectionOwner: int -> Drawable -> int -> unit
val XGetSelectionOwner: int -> Drawable

val XConvertSelection: { selection: int,
                         target:    int,
                         property:  int,
                         requestor: Drawable,
                         time:      int } -> unit

val XSendSelectionNotify: { selection: int,
                            target:    int,
                            property:  int,
                            requestor: Drawable,
                            time:      int } -> unit
```

Syntax:

```
XSetSelectionOwner selection owner time ;
val owner = XGetSelectionOwner selection ;
XConvertSelection    {selection,target,property,requestor,time} ;
XSendSelectionNotify {selection,target,property,requestor,time} ;
```

Arguments:

selection	Specifies the selection atom.
owner	Specifies/returns the owner of the specified selection atom. You can pass a window or **NoDrawable**.
time	Specifies the time. You can pass either a timestamp or **CurrentTime**.
target	Specifies the target atom.
property	Specifies the property name. You also can pass zero.
requestor	Specifies the requestor.

Argument Type:

```
val CurrentTime: int
```

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 **NoDrawable**, then the owner of the selection becomes **NoDrawable** (that is, no owner). Otherwise, the owner of the selection becomes the client executing the request.

If the new owner (whether a client or **NoDrawable**) is not the same as the current owner of the selection and the current owner is not **NoDrawable**, 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 **NoDrawable**, 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.

The **XGetSelectionOwner** function returns the **Drawable** associated with the window that currently owns the specified selection. If no selection was specified, the function returns the constant **NoDrawable**. If **NoDrawable** is returned, there is no owner for the selection.

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 zero.

The arguments are passed on unchanged in either of the events. There are two predefined selection atoms: **XA_PRIMARY** and **XA_SECONDARY**.

XSendSelectionNotify is called when you have received a **SelectionRequest** event asking for the selection, which you currently own, to be converted to some desired type. When you have completed the conversion you store the converted value in the indicated property on the window. Then you call **XSendSelectionNotify** with the same parameters as the **SelectionRequest** event to indicate that the conversion was successful. If cannot

perform the conversion then you call **XSendSelectionNotify** with property set to zero to indicate that the conversion failed. If the conversion was successful then the requestor will read the value from the property on the window, and will delete the property to indicate that the transfer has been completed.

2.13 Screen Saver

2.13.1 XSetScreenSaver, XForceScreenSaver, XActivateScreenSaver, XResetScreenSaver, XGetScreenSaver

Types:

```
val XSetScreenSaver:      int -> int -> Blanking -> Exposures -> unit
val XForceScreenSaver:    SaveMode -> unit
val XActivateScreenSaver: unit -> unit
val XResetScreenSaver:    unit -> unit
val XGetScreenSaver:      unit -> (int * int * Blanking * Exposures)
```

Syntax:

```
XSetScreenSaver timeout interval preferBlanking allowExposures ;
XForceScreenSaver mode ;
XActivateScreenSaver() ;
XResetScreenSaver() ;
val (timeout,interval,preferBlanking,allowExposures) = XGetScreenSaver() ;
```

Arguments:

allowExposures	Specifies/returns the screen save control values. You can pass **DontAllowExposures**, **AllowExposures**, or **DefaultExposures**.
interval	Specifies/returns the interval, in seconds, between screen saver alterations.
mode	Specifies the mode that is to be applied. You can pass **ScreenSaverActive** or **ScreenSaverReset**.
preferBlanking	Specifies/returns how to enable screen blanking. You can pass **DontPreferBlanking**, **PreferBlanking**, or **DefaultBlanking**.
timeout	Specifies the timeout, in seconds, until the screen saver turns on.

Argument Type:

```
datatype SaveMode  = ScreenSaverReset   | ScreenSaverActive
datatype Blanking  = DontPreferBlanking | PreferBlanking | DefaultBlanking
datatype Exposures = DontAllowExposures | AllowExposures | DefaultExposures
```

Description:

Timeout and interval are specified in seconds. A timeout of 0 disables the screen saver (but an activated screen saver is not deactivated), and a timeout of ˜1 restores the default. Other negative values generate a **BadValue** error. If the timeout value is non-zero,

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.

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).

The **XActivateScreenSaver** function activates the screen saver.

The **XResetScreenSaver** function resets the screen saver.

The **XGetScreenSaver** function gets the current screen saver values.

2.14 Tiles, Stipples, Bitmaps and Pixmaps

2.14.1 XCreatePixmap, XFreePixmap

Types:

```
val XCreatePixmap: Drawable -> XRectangle -> int -> Drawable
val XFreePixmap:   Drawable -> unit
```

Syntax:

```
val pixmap = XCreatePixmap d area depth ;
XFreePixmap pixmap ;
```

Arguments:

d	Specifies which screen the pixmap is created on.
depth	Specifies the depth of the pixmap.
pixmap	Specifies the pixmap.
area	Specifies 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 **Drawable** that identifies it. It is valid to pass an **InputOnlyClass** window to the drawable argument. The width and height arguments must be non-zero, or a **BadValue** error results. The depth argument must be one of the depths supported by the screen of the specified drawable, or a **BadValue** error results.

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

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

2.14.2 XReadBitmapFile, XWriteBitmapFile, XCreatePixmapFromBitmapData, XCreateBitmapFromData

Types:

```
val XReadBitmapFile: Drawable -> string -> BitmapStatus

val XWriteBitmapFile: string -> Drawable ->
                      XRectangle -> XPoint -> BitmapStatus

val XCreatePixmapFromBitmapData: Drawable -> string -> XRectangle ->
                                 int -> int -> int -> Drawable

val XCreateBitmapFromData: Drawable -> string -> XRectangle -> Drawable
```

Syntax:

```
val status = XReadBitmapFile d filename ;
val status = XWriteBitmapFile filename bitmap area hotspot ;
val pixmap = XCreatePixmapFromBitmapData d data area fg bg depth ;
val bitmap = XCreateBitmapFromData d data area ;
```

Arguments:

bitmap	Specifies the bitmap.
status	Returns the bitmap that is created, or an error condition.
d	Specifies the drawable that indicates the screen.
data	Specifies the data in bitmap format.
depth	Specifies the depth of the pixmap.
fg	Specifies the foreground and
bg	background pixel values to use.
filename	Specifies the file name to use.
area	Specifies the width and height.

hotspot Specifies where to place the hotspot coordinates, or (~1,~1) if none are present in the file.

Argument Type:

```
datatype BitmapStatus = BitmapOpenFailed
                      | BitmapFileInvalid
                      | BitmapNoMemory
                      | BitmapSuccess of Drawable * XRectangle * XPoint
```

Description:

The **XReadBitmapFile** function reads in a file containing a bitmap. The ability to read other than the standard format is implementation dependent. If the file cannot be opened, **XReadBitmapFile** returns **BitmapOpenFailed**. If the file can be opened but does not contain valid bitmap data, it returns **BitmapFileInvalid**. If insufficient working storage is allocated, it returns **BitmapNoMemory**. If the file is readable and valid, it returns **BitmapSuccess**.

XReadBitmapFile reads the bitmap's height and width from the file. It then creates a pixmap of the appropriate size and reads the bitmap data from the file into the pixmap. The caller must free the bitmap using **XFreePixmap** when finished. If the hotspot is defined in the bitmap file, **XReadBitmapFile** returns the hotspot in the status as well, otherwise it returns (~1,~1).

The **XWriteBitmapFile** function writes a bitmap out to a file in the 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 the hotspot is not (~1,~1), **XWriteBitmapFile** writes it out as the hotspot coordinates for the bitmap.

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.

The **XCreateBitmapFromData** function allows you to include a bitmap file without reading in the bitmap file. The following example creates a weave bitmap:

```
val data = [17,  17, 184, 184, 124, 124, 58,  58,
            17,  17, 163, 163, 199, 199, 139, 139,
            17,  17, 184, 184, 124, 124, 58,  58,
            17,  17, 163, 163, 199, 199, 139, 139] ;

fun MakeData []     = ""
|   MakeData (H::T) = chr H ^ MakeData T ;

val wideWeave = XCreateBitmapFromData root (MakeData data)
                                           (Area{x=0,y=0,w=16,h=16}) ;
```

If insufficient working storage was allocated, **XCreateBitmapFromData** returns **NoDrawable**. It is your responsibility to free the bitmap using **XFreePixmap** when finished.

2.15 User Preferences

2.15.1 XAutoRepeatOn, XAutoRepeatOff, XBell, XQueryKeymap

Types:

```
val XAutoRepeatOff: unit -> unit
val XAutoRepeatOn:  unit -> unit
val XBell:          int  -> unit
val XQueryKeymap:   unit -> bool list (* 256 bools *)
```

Syntax:

```
XAutoRepeatOn() ;
XAutoRepeatOff() ;
XBell percent ;
val keymap = XQueryKeymap() ;
```

Arguments:

percent	Specifies the volume for the bell, which can range from ˜100 to 100 inclusive.
keymap	Returns the keyboard state vector

Description:

The **XAutoRepeatOn** function turns on auto-repeat for the keyboard.

The **XAutoRepeatOff** function turns off auto-repeat for the keyboard.

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.

The **XQueryKeymap** function returns a bit vector for the logical state of the keyboard. The vector is returned as a list of 256 bools, representing the keys 0 to 255 in that order. Each bool set to true indicates that the corresponding key is currently pressed.

Note that the logical state of a device (as seen by client applications) may lag the physical state if device event processing is frozen.

2.15.2 XGetDefault

Types:

```
val XGetDefault: string -> string -> string
```

Syntax:

```
val default = XGetDefault program option ;
```

Arguments:

option	Specifies the option name.
program	Specifies the program name.

Description:

XGetDefault returns the value for the program and option entry in the user's defaults database. If **XGetDefault** fails then exception **XWindows** is raised with "XGetDefault failed" .

2.16 Windows

2.16.1 XCreateWindow, XCreateSimpleWindow

Types:

```
val XCreateWindow: Drawable -> XPoint -> XRectangle ->
                   int -> int -> WindowClass -> Visual ->
                   XSetWindowAttributes list -> Drawable

val XCreateSimpleWindow: Drawable -> XPoint -> XRectangle ->
                         int -> int -> int -> Drawable
```

Syntax:

```
val window = XCreateWindow parent point area
             borderWidth depth class visual attributes ;

val window = XCreateSimpleWindow parent point area
             borderWidth borderPixel backgroundPixel ;
```

Arguments:

attributes	Specifies the initial values for the window's attributes.
backgroundPixel	Specifies the background pixel value of the window.
borderPixel	Specifies the border pixel value of the window.
borderWidth	Specifies the width of the window's border in pixels.
class	Specifies the window's class. You can pass **InputOutputClass**, **InputOnlyClass**, or **CopyFromParentClass**. A class of **CopyFromParentClass** means the class is taken from the parent.
depth	Specifies the window's depth. A depth of zero means the depth is taken from the parent.
parent	Specifies the parent window.

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

area Specifies the width and height, which are the created window's inside dimensions and do not include the created window's borders.

point Specifies 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.

Argument Type:

```
datatype XSetWindowAttributes = CWBackPixmap       of Drawable
                              | CWBackPixel        of int
                              | CWBorderPixmap     of Drawable
                              | CWBorderPixel      of int
                              | CWBitGravity       of Gravity
                              | CWWinGravity       of Gravity
                              | CWBackingStore     of BackingStore
                              | CWBackingPlanes    of int
                              | CWBackingPixel     of int
                              | CWOverrideRedirect of bool
                              | CWSaveUnder        of bool
                              | CWEventMask        of EventMask list
                              | CWDontPropagate    of EventMask list
                              | CWColormap         of Colormap
                              | CWCursor           of Cursor

datatype BackingStore = NotUseful | WhenMapped | Always

datatype WindowClass = CopyFromParentClass
                     | InputOutputClass
                     | InputOnlyClass
```

Description:

The **XCreateWindow** function creates an unmapped subwindow for a specified parent window, returns the **Drawable** value for 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 borderWidth for an **InputOnlyClass** window must be zero, or a **BadMatch** error results. For class **InputOutputClass**, 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 **InputOnlyClass**, or a **BadMatch** error results. For an **InputOnlyClass** 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.

If **XCreateWindow** fails then exception **XWindows** is raised with "XCreateWindow failed" .

The **XCreateSimpleWindow** function creates an unmapped **InputOutputClass** subwindow for a specified parent window, returns the **Drawable** value for 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 borderWidth for an **InputOnlyClass** 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. If **XCreateSimpleWindow** fails then exception **XWindows** is raised with "XCreateSimpleWindow failed" .

2.16.2 XDestroyWindow, XDestroySubwindows

Types:

```
val XDestroyWindow:     Drawable -> unit
val XDestroySubwindows: Drawable -> unit
```

Syntax:

```
XDestroyWindow w ;
XDestroySubwindows w ;
```

Arguments:

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.

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. If **XDestroySubwindows** fails then exception **XWindows** is raised with "XDestroySubwindows failed" .

2.16.3 XGetGeometry, XGetWindowAttributes

Types:

```
val XGetGeometry: Drawable -> (Drawable * XPoint * XRectangle * int * int)

val XGetWindowAttributes: Drawable -> XWindowAttributes
```

Syntax:

```
val (root,position,size,borderWidth,depth) = XGetGeometry w ;
val attributes = XGetWindowAttributes w ;
```

Arguments:

d	Specifies the drawable, which can be a window or a pixmap.
root	Returns the root window
position	Returns 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.
size	Returns the drawable's dimensions (width and height).
borderWidth	Returns the border width in pixels.
depth	Returns the depth of the drawable (bits per pixel for the object).

Argument Type:

```
datatype WindowClass = CopyFromParentClass
                     | InputOutputClass
                     | InputOnlyClass

datatype MapState = IsUnmapped | IsUnviewable | IsViewable

datatype Gravity = ForgetGravity    | NorthWestGravity | NorthGravity
                 | NorthEastGravity | WestGravity      | CenterGravity
                 | EastGravity      | SouthWestGravity | SouthGravity
                 | SouthEastGravity | StaticGravity

val UnmapGravity: Gravity    (* same as ForgetGravity *)

datatype BackingStore = NotUseful | WhenMapped | Always

val NoColormap: Colormap

datatype XWindowAttributes = XWindowAttributes of
                             {
                               position:          XPoint,
                               size:              XRectangle,
                               borderWidth:       int,
                               depth:             int,
                               visual:            Visual,
                               root:              Drawable,
                               class:             WindowClass,
                               bitGravity:        Gravity,
                               winGravity:        Gravity,
                               backingStore:      BackingStore,
                               backingPlanes:     int,
                               backingPixel:      int,
```

```
  saveUnder:          bool,
  colormap:           Colormap,
  mapInstalled:       bool,
  mapState:           MapState,
  allEventMasks:      EventMask list,
  yourEventMask:      EventMask list,
  doNotPropagateMask: EventMask list,
  overrideRedirect:   bool
}
```

Argument Description:

The position member is set to the upper-left outer corner relative to the parent window's origin. The size member is set to the inside size of the window, not including the border. The borderWidth 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 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 **InputOutputClass** or **InputOnlyClass**.

The bitGravity 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 winGravity 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
```

The backingStore member is set to indicate how the X server should maintain the contents of a window and can be **WhenMapped**, **Always**, or **NotUseful**. The backingPlanes 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 backingPixel member is set to indicate what values to use for planes not set in backingPlanes.

The saveUnder member is set to true or false. The colormap member is set to the colormap for the specified window and can be a **Colormap** or **NoColormap**. The mapInstalled member is set to indicate whether the colormap is currently installed and can be true or false. The mapState 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 allEventMasks member is set to the event masks selected on the window by all clients. The yourEventMask member is set to the event masks selected by the querying client. The doNotPropagateMask member is set to the list of events that should not propagate.

The overrideRedirect 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.

Description:

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

If **XGetWindowAttributes** fails then exception **XWindows** is raised with "XGetWindowAttributes failed" .

The **XGetGeometry** function returns the root window and the current geometry of the drawable. The geometry of the drawable includes the position as x and y coordinates, the size as width and height, the border width, and the depth. These are described in the argument list. It is legal to pass to this function a window whose class is **InputOnlyClass**. If **XGetGeometry** fails then exception **XWindows** is raised with "XGetGeometry failed" .

2.16.4 XGetWindowRoot, XGetWindowPosition, XGetWindowSize, XGetWindowBorderWidth, XGetWindowDepth, XGetWindowParent, XGetWindowChildren

Types:

```
val XGetWindowRoot:        Drawable -> Drawable
val XGetWindowPosition:    Drawable -> XPoint
val XGetWindowSize:        Drawable -> XRectangle
val XGetWindowBorderWidth: Drawable -> int
val XGetWindowDepth:       Drawable -> int
val XGetWindowParent:      Drawable -> Drawable
val XGetWindowChildren:    Drawable -> Drawable list
```

Description:

These convenience functions return the individual attributes returned in bulk by **XGetGeometry** and **XQueryTree**.

XGetWindowRoot returns the root window for the drawable. **XGetWindowPosition** returns the coordinates of the outer top left corner of the window. **XGetWindowSize** returns the inside size of the window. **XGetWindowBorderWidth** returns the border width in pixels of the window. **XGetWindowDepth** returns the depth of the window. **XGetWindowParent** returns the parent window of the specified window. **XGetWindowChildren** returns the children of the specified window.

2.16.5 XChangeWindowAttributes, XSetWindowBackground, XSetWindowBackgroundPixmap, XSetWindowBorder, XSetWindowBorderPixmap

Types:

```
val XChangeWindowAttributes:    Drawable -> XSetWindowAttributes list -> unit
val XSetWindowBackground:       Drawable -> int      -> unit
val XSetWindowBackgroundPixmap: Drawable -> Drawable -> unit
val XSetWindowBorder:           Drawable -> int      -> unit
val XSetWindowBorderPixmap:     Drawable -> Drawable -> unit
```

Syntax:

```
XChangeWindowAttributes w attributes ;
XSetWindowBackground w backgroundPixel ;
XSetWindowBackgroundPixmap w backgroundPixmap ;
XSetWindowBorder w borderPixel ;
XSetWindowBorderPixmap w borderPixmap ;
```

Arguments:

attributes	Specifies the list of attributes to change.
backgroundPixel	Specifies the pixel that is to be used for the background.
backgroundPixmap	Specifies the background pixmap, **ParentRelative**, or **NoDrawable**.
borderPixel	Specifies the entry in the colormap.
borderPixmap	Specifies the border pixmap or **CopyFromParentDrawable**.
w	Specifies the window.

Argument Type:

```
datatype XSetWindowAttributes = CWBackPixmap       of Drawable
                              | CWBackPixel        of int
                              | CWBorderPixmap     of Drawable
                              | CWBorderPixel      of int
                              | CWBitGravity       of Gravity
                              | CWWinGravity       of Gravity
                              | CWBackingStore     of BackingStore
                              | CWBackingPlanes    of int
                              | CWBackingPixel     of int
                              | CWOverrideRedirect of bool
                              | CWSaveUnder        of bool
                              | CWEventMask        of EventMask list
                              | CWDontPropagate    of EventMask list
                              | CWColormap         of Colormap
                              | CWCursor           of Cursor
```

Description:

The **XChangeWindowAttributes** function uses the window attributes in the **XSetWindowAttributes** list 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 **NoDrawable** or **ParentRelative** restores the default background pixmap. Changing the border of a root window to **CopyFromParentDrawable** 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 **NoCursor** 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.

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 **InputOnlyClass** window, a **BadMatch** error results.

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 **InputOnlyClass** window, a **BadMatch** error results. If the background is set to **NoDrawable**, the window has no defined background.

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

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 **CopyFromParentDrawable**, a copy of the parent window's border pixmap is used. If you attempt to perform this on an **InputOnlyClass** window, a **BadMatch** error results.

2.16.6 XConfigureWindow, XMoveWindow, XResizeWindow, XMoveResizeWindow, XSetWindowBorderWidth

Types:

```
val XConfigureWindow:      Drawable -> XWindowChanges list -> unit
val XMoveWindow:           Drawable -> XPoint -> unit
val XResizeWindow:         Drawable -> XRectangle -> unit
val XMoveResizeWindow:     Drawable -> XPoint -> XRectangle -> unit
val XSetWindowBorderWidth: Drawable -> int -> unit
```

Syntax:

```
XConfigureWindow w changes ;
XMoveWindow w origin ;
XResizeWindow w area ;
XMoveResizeWindow w origin area ;
XSetWindowBorderWidth w borderWidth ;
```

Arguments:

changes	Specifies a list of **XWindowChanges**.
w	Specifies the window to be reconfigured.
borderWidth	Specifies the width of the window border.
area	Specifies the interior dimensions of the window.
origin	Specifies 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 relative to its parent.

Argument Type:

```
datatype XWindowChanges = CWPosition    of XPoint
                        | CWSize        of XRectangle
                        | CWBorderWidth of int
                        | CWStackMode   of StackMode
                        | CWSibling     of Drawable

datatype StackMode = Above | Below | TopIf | BottomIf | Opposite
```

Argument Description:

The **CWPosition** member is 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 **CWSize** member is used to set the inside size of the window, not including the border, and must be non-zero, or a **BadValue** error results. Attempts to configure a root window have no effect.

The **CWBorderWidth** 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 **InputOnlyClass** window non-zero, a **BadMatch** error results.

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

Description:

The **XConfigureWindow** function uses the values specified in the **XWindowChanges** list 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).

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.

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.

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.

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

2.16.7 XMapWindow, XMapRaised, XMapSubwindows

Types:

```
val XMapWindow:     Drawable -> unit
val XMapRaised:     Drawable -> unit
val XMapSubwindows: Drawable -> unit
```

Syntax:

```
XMapWindow w ;
XMapRaised w ;
XMapSubwindows w ;
```

Arguments:

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 **InputOutputClass** window, **XMapWindow** generates **Expose** events on each **InputOutputClass** 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.

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.

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.

2.16.8 XQueryPointer

Types:

```
val XQueryPointer: Drawable -> (bool *
                                Drawable * Drawable *
                                XPoint * XPoint * Modifier list)
```

Syntax:

```
val (sameScreen,root,child,rootPointer,pointer,modifiers) = XQueryPointer w ;
```

Arguments:

sameScreen	Returns true if the pointer is on the same screen as the specified window.
child	Returns the child window that the pointer is located in, if any.
modifiers	Returns the current state of the modifier keys and pointer buttons.
root	Returns the root window that the pointer is in.
rootPointer	Return the pointer coordinates relative to the root window's origin.
w	Specifies the window.

pointer 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 sameScreen is false, the pointer is not on the same screen as the specified window, and **XQueryPointer** returns **NoDrawable** to child and (0,0) to pointer. If sameScreen is true, the pointer coordinates returned to pointer are relative to the origin of the specified window. In this case, **XQueryPointer** returns the child that contains the pointer, if any, or else **NoDrawable** to child.

XQueryPointer returns the current logical state of the keyboard buttons and the modifier keys in modifiers. It sets modifiers to the list of button or modifier key masks to match the current state of the mouse buttons and the modifier keys.

2.16.9 XQueryTree

Types:

```
val XQueryTree: Drawable -> (Drawable * Drawable * Drawable list)
```

Syntax:

```
val (root,parent,children) = XQueryTree w ;
```

Arguments:

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

Description:

The **XQueryTree** function returns the root window, the parent window, and a list of children windows for the specified window. The children are listed in current stacking order, from bottom-most (first) to top-most (last). If it fails, **XQueryTree** raises exception **XWindows** with "XQueryTree failed" .

2.16.10 XRaiseWindow, XLowerWindow, XCirculateSubwindows, XCirculateSubwindowsDown, XCirculateSubwindowsUp, XRestackWindows

Types:

```
val XRaiseWindow:             Drawable -> unit
val XLowerWindow:             Drawable -> unit
val XCirculateSubwindows:     Drawable -> CirculateDirection -> unit
val XCirculateSubwindowsDown: Drawable -> unit
val XCirculateSubwindowsUp:   Drawable -> unit
val XRestackWindows:          Drawable list -> unit
```

Syntax:

```
XRaiseWindow w ;
XLowerWindow w ;
XCirculateSubwindows w direction ;
XCirculateSubwindowsDown w ;
XCirculateSubwindowsUp w ;
XRestackWindows windows ;
```

Arguments:

direction	Specifies the direction (up or down) that you want to circulate the window. You can pass **RaiseLowest** or **LowerHighest**.
w	Specifies the window.
windows	Specifies the list of windows to be restacked.

Argument Type:

```
datatype CirculateDirection = RaiseLowest | LowerHighest
```

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.

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.

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.

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.

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.

The **XRestackWindows** function restacks the windows in the order specified, from top to bottom. The stacking order of the first window in the windows list is unaffected, but the other windows in the list are stacked underneath the first window, in the order of the list. The stacking order of the other windows is not affected. For each window in the window list 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.

2.16.11 XReparentWindow

Types:

```
val XReparentWindow: Drawable -> Drawable -> XPoint -> unit
```

Syntax:

```
XReparentWindow w parent topLeft ;
```

Arguments:

parent	Specifies the parent window.
w	Specifies the window.
topLeft	Specifies the x and y coordinates of 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 overrideRedirect 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, or if the new parent window is the specified window or an inferior of the specified window, or if the specified window has a **ParentRelative** background, and the new parent window is not the same depth as the specified window.

2.16.12 XUnmapWindow, XUnmapSubwindows

Types:

```
val XUnmapWindow:     Drawable -> unit
val XUnmapSubwindows: Drawable -> unit
```

Syntax:

```
XUnmapWindow w ;
XUnmapSubwindows w ;
```

Arguments:

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.

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.

2.17 Window Manager

2.17.1 XSetIconSizes, XGetIconSizes

Types:

```
val XSetIconSizes: Drawable ->
                   (XRectangle * XRectangle * XRectangle) list -> unit

val XGetIconSizes: Drawable ->
                   (XRectangle * XRectangle * XRectangle) list
```

Syntax:

```
XSetIconSizes w sizes ;
val sizes = XGetIconSizes w ;
```

Arguments:

sizes	Specifies the size list.
w	Specifies the window.

Description:

The **XSetIconSizes** function is used only by window managers to set the supported icon sizes. The size is specified as (minimum size,maximum size,size increment).

The **XGetIconSizes** function returns the empty list if a window manager has not set icon sizes, otherwise it returns a list of supported sizes. **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.

2.17.2 XSetTransientForHint, XGetTransientForHint

Types:

```
val XSetTransientForHint: Drawable -> Drawable -> unit
val XGetTransientForHint: Drawable -> Drawable
```

Syntax:

```
XSetTransientForHint transientWindow mainWindow ;
val mainWindow = XGetTransientForHint transientWindow ;
```

Arguments:

transientWindow	Specifies the transient window.
mainWindow	Specifies a more permanent window in the application.

Properties:

WM_TRANSIENT_FOR	Set by application programs to indicate to the window manager a transient top-level window, such as a dialog box.

Description:

The **XSetTransientForHint** function sets the **WM_TRANSIENT_FOR** property of transientWindow to mainWindow.

The **XGetTransientForHint** function returns the **WM_TRANSIENT_FOR** property for the specified transientWindow. If the property does not exist then exception **XWindows** is raised with "XGetTransientForHint failed" .

2.17.3 XSetWMClass, XGetWMClass

Types:

```
val XSetWMClass: Drawable -> string list -> unit
val XGetWMClass: Drawable -> string list
```

Syntax:

```
XSetWMClass w class ;
val class = XGetWMClass w ;
```

Arguments:

class	Specifies the class names for the window
w	Specifies the window

Properties:

WM_CLASS	Set by application programs to allow window and session managers to obtain the application's resources from the resource database.

Description:

XSetWMClass sets the **WM_CLASS** property on the specified window. **XGetWM-Class** returns the **WM_CLASS** property on the specified window.

2.17.4 XSetWMClientMachine, XGetWMClientMachine

Types:

```
val XSetWMClientMachine: Drawable -> string -> unit
val XGetWMClientMachine: Drawable -> string
```

Syntax:

```
XSetWMClientMachine w machine ;
val machine = XGetWMClientMachine w ;
```

Arguments:

w	Specifies the window
machine	Specifies the name of the machine on which the application is running.

Properties:

WM_CLIENT_MACHINE	The string name of the machine on which the client application is running.

Description:

The **XSetWMClientMachine** convenience function performs a **XSetProperty** on the **WM_CLIENT_MACHINE** property.

The **XGetWMClientMachine** convenience function performs an **XGetTextProperty** on the **WM_CLIENT_MACHINE** property.

2.17.5 XSetWMColormapWindows, XGetWMColormapWindows

Types:

```
val XSetWMColormapWindows: Drawable -> Drawable list -> unit
val XGetWMColormapWindows: Drawable -> Drawable list
```

Syntax:

```
XSetWMColormapWindows topWindow colormapWindows ;
val colormapWindows = XGetWMColormapWindows topWindow ;
```

Arguments:

colormapWindows	Specifies the list of windows.
topWindow	Specifies one of the application's top level windows.

Properties:

WM_COLORMAP_WINDOWS	List of windows that may need a different colormap than that of their top-level window.

Description:

The **XSetWMColormapWindows** function replaces the **WM_COLORMAP_WINDOWS** property on the specified window with the list of windows specified by the colormapWindows argument. The property is stored with a type of **XA_WINDOW** and a format of 32. If it cannot intern the **WM_COLORMAP_WINDOWS** atom, **XSetWMColormapWindows** raises exception **XWindows** with "XSetWMColormapWindows failed" .

The **XGetWMColormapWindows** function returns the list of window identifiers stored in the **WM_COLORMAP_WINDOWS** property on the specified window. These identifiers indicate the colormaps that the window manager may need to install for this window. If the property exists, is of type WINDOW, is of format 32, and the atom **WM_COLORMAP_WINDOWS** can be interned, **XGetWMColormapWindows** returns the list of windows. Otherwise, it returns the empty list.

2.17.6 XSetWMCommand, XGetWMCommand

Types:

```
val XSetWMCommand: Drawable -> string list -> unit
val XGetWMCommand: Drawable -> string list
```

Syntax:

```
XSetWMCommand w commands ;
val commands = XGetWMCommand w ;
```

Arguments:

w	Specifies the window.
commands	Specifies the list of strings.

Description:

The **XSetWMCommand** function sets the **WM_COMMAND** property on the specified window. Typically it is set to the command and arguments used to invoke the application.

The **XGetWMCommand** function reads the **WM_COMMAND** property from the specified window and returns a string list. If the **WM_COMMAND** property exists, and it is of type **XA_STRING** and format 8 then it is returned as a string list. Otherwise, it raises exception **XWindows** with "XGetWMCommand" .

2.17.7 XSetWMHints, XGetWMHints

Types:

```
val XSetWMHints: Drawable -> XWMHint list -> unit
val XGetWMHints: Drawable -> XWMHint list
```

Syntax:

```
XSetWMHints w hints ;
val hints = XGetWMHints w ;
```

Arguments:

w	Specifies the window
hints	Specifies the list of **XWMHint** values

Argument Type:

```
datatype XWMStateHint = DontCareState | NormalState | ZoomState
                      | IconicState   | InactiveState

datatype XWMHint = InputHint        of bool
                 | StateHint        of XWMStateHint
                 | IconPixmapHint   of Drawable
                 | IconWindowHint   of Drawable
                 | IconPositionHint of XPoint
                 | IconMaskHint     of Drawable
```

Argument Description:

The **InputHint** 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).

Possible values for the **StateHint** member are

DontCareState	(* don't know or care *)
NormalState	(* most applications want to start this way *)
ZoomState	(* application wants to start zoomed *)
IconicState	(* application wants to start as an icon *)
InactiveState	(* application wants to start invisibly *)

The **IconMaskHint** member specifies which pixels of the **IconPixmapHint** member should be used as the icon. This allows for nonrectangular icons. Both the **IconPixmapHint** member and the **IconMaskHint** member must be bitmaps. The **IconWindowHint** member lets an application provide a window for use as an icon for window managers that support such use. The **IconPositionHint** member specifies a position on the screen for the icon.

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.

The **XGetWMHints** function reads the window manager hints and returns the empty list if no **WM_HINTS** property was set on the window or returns a list of XWMHints if it succeeds.

2.17.8 XSetWMIconName, XGetWMIconName

Types:

```
val XSetWMIconName: Drawable -> string -> unit
val XGetWMIconName: Drawable -> string
```

Syntax:

```
XSetWMIconName w iconName ;
val iconName = XGetWMIconName w ;
```

Arguments:

iconName	Specifies the icon name
w	Specifies the window

Description:

The **XSetWMIconName** convenience function performs a **XSetProperty** on the **WM_ICON_NAME** property. The **XSetWMIconName** function sets the name to be displayed in a window's icon.

The **XGetWMIconName** convenience function performs an **XGetTextProperty** on the **WM_ICON_NAME** property. The **XGetWMIconName** function returns the name to be displayed in the specified window's icon. If it succeeds, it returns the name, otherwise, if no icon name has been set for the window, it raises exception **XWindows** with "XGetWMIconName" .

2.17.9 XSetWMName, XGetWMName

Types:

```
val XSetWMName: Drawable -> string -> unit
val XGetWMName: Drawable -> string
```

Syntax:

```
XSetWMName w windowName ;
windowName = XGetWMName w ;
```

Arguments:

windowName	Specifies the window name
w	Specifies the window

Description:

The **XSetWMName** convenience function performs a **XSetProperty** on the **WM_NAME** property. The **XSetWMName** function assigns the name passed to windowName 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.

The **XGetWMName** convenience function performs an **XGetTextProperty** on the **WM_NAME** property. The **XGetWMName** function returns the name of the specified window. If the **WM_NAME** property has not been set for this window, **XGetWMName** raises exception **XWindows** with "XGetWMName" .

2.17.10 XSetWMProperties

Types:

```
val XSetWMProperties: Drawable ->
                      string -> string -> string list ->
                      XWMSizeHint list -> XWMHint list ->
                      string list -> unit
```

Syntax:

```
XSetWMProperties w windowName iconName commands normalHints wmHints class ;
```

Arguments:

w	Specifies the window
windowName	Specifies the window name
iconName	Specifies the icon name
commands	Specifies the list of strings used to invoke the window
normalHints	Specifies the list of **XWMSizeHint** values for the window
wmHints	Specifies the list of **XWMHint** values for the window
class	Specifies the class names for the window

Properties:

WM_CLASS	Set by application programs to allow window and session managers to obtain the application's resources from the resource database.
WM_CLIENT_MACHINE	The string name of the machine on which the client application is running.
WM_COMMAND	The command and arguments, separated by ASCII nulls, used to invoke the application.
WM_HINTS	Additional hints set by client for use by the window manager. The ML type of this property is XWMHints.
WM_ICON_NAME	Name to be used in icon.
WM_NAME	Name of the application.
WM_NORMAL_HINTS	Size hints for a window in its normal state. The ML type of this property is **XWMSizeHint**.

Description:

The **XSetWMProperties** convenience function provides a single programming interface for setting those essential window properties that are used for communicating with other clients (particularly window and session managers).

If the windowName argument is not empty, **XSetWMProperties** calls **XSetWMName**, which, in turn, sets the **WM_NAME** property. If the iconName argument is not empty, **XSetWMProperties** calls **XSetWMIconName**, which sets the **WM_ICON_NAME** property. If the commands argument is not empty, **XSetWMProperties** calls **XSetWMCommand**, which sets the **WM_COMMAND** property.

If the normalHints argument is not empty, **XSetWMProperties** calls **XSetWMNormalHints**, which sets the **WM_NORMAL_HINTS** property. If the wmHints argument is not empty, **XSetWMProperties** calls **XSetWMHints**, which sets the **WM_HINTS** property. If the class argument is not empty, **XSetWMProperties** calls **XSetWMClass**, which sets the **WM_CLASS** property.

2.17.11 XSetWMSizeHints, XGetWMSizeHints, XSetWMNormalHints, XGetWMNormalHints

Types:

```
val XSetWMSizeHints: Drawable -> int -> XWMSizeHint list -> unit
val XGetWMSizeHints: Drawable -> int -> XWMSizeHint list
val XSetWMNormalHints: Drawable -> XWMSizeHint list -> unit
val XGetWMNormalHints: Drawable -> XWMSizeHint list
```

Syntax:

```
XSetWMSizeHints w property hints ;
val hints = XGetWMSizeHints w property ;
XSetWMNormalHints w hints ;
val hints = XGetWMNormalHints w ;
```

Arguments:

w	Specifies the window
property	Specifies the property atom
hints	Specifies the list of **XWMSizeHint** values

Argument Type:

```
datatype XWMSizeHint = PPosition   of XPoint
                     | PSize       of XRectangle
                     | PMinSize    of XRectangle
                     | PMaxSize    of XRectangle
                     | PResizeInc  of XRectangle
                     | PAspect     of XPoint * XPoint
                     | PBaseSize   of XRectangle
                     | PWinGravity of Gravity
```

Argument Description:

The **PPosition** and **PSize** members are now obsolete and are left solely for compatibility reasons. The **PMinSize** member specifies the minimum window size that still allows the application to be useful. The **PMaxSize** member specifies the maximum window size. The **PResizeInc** member defines a size increment which the window prefers to be resized to. The two points in the **PAspect** member give minimum and maximum aspect ratios. They are expressed as ratios of x and y, and they allow an application to specify the range of aspect ratios it prefers. The **PBaseSize** member defines the desired size of the window. The **PWinGravity** member defines the region of the window that is to be retained when it is resized.

Description:

The **XSetWMSizeHints** function replaces the size hints for the specified property on the named window. If the specified property does not already exist, **XSetWMSizeHints** sets the size hints for the specified property on the named window. The property is stored with a type of **WM_SIZE_HINTS** and a format of 32. To set a window's normal size hints, you can use the **XSetWMNormalHints** function.

The **XGetWMSizeHints** function returns the size hints stored in the specified property on the named window. If the property is of type **WM_SIZE_HINTS**, of format 32, and is long enough to contain either an old (pre-ICCCM) or new size hints structure, **XGetWMSizeHints** returns the list of fields that were supplied by the user. Otherwise, it returns the empty list. To get a window's normal size hints, you can use the **XGetWMNormalHints** function.

If **XGetWMSizeHints** returns successfully and a pre-ICCCM size hints property is read, the list returned may contain the following members:

```
[PPosition,PSize,PMinSize,PMaxSize,PResizeInc,PAspect]
```

If the property is large enough to contain the base size and window gravity fields as well, the list returned may contain the following members:

```
[PBaseSize,PWinGravity]
```

The **XSetWMNormalHints** function replaces the size hints for the **WM_NORMAL_HINTS** property on the specified window. If the property does not already exist, **XSetWMNormalHints** sets the size hints for the **WM_NORMAL_HINTS** property on the specified window. The property is stored with a type of **WM_SIZE_HINTS** and a format of 32.

The **XGetWMNormalHints** function returns the size hints stored in the **WM_NORMAL_HINTS** property on the specified window. If the property is of type **WM_SIZE_HINTS**, of format 32, and is long enough to contain either an old (pre-ICCCM) or new size hints structure, **XGetWMNormalHints** returns the list of fields that were supplied by the user. Otherwise, it returns the empty list.

If **XGetWMNormalHints** returns successfully and a pre-ICCCM size hints property is read, the list returned may contain the following members:

```
[PPosition,PSize,PMinSize,PMaxSize,PResizeInc,PAspect]
```

If the property is large enough to contain the base size and window gravity fields as well, the list returned may contain the following members:

```
[PBaseSize,PWinGravity]
```

2.17.12 XWMGeometry

Types:

```
val XWMGeometry: string -> string -> int ->
                 XWMSizeHint list -> XPoint * XRectangle * Gravity
```

Syntax:

```
val (topLeft,area,gravity) = XWMGeometry userGeometry
                                         defaultGeometry
                                         borderWidth
                                         sizeHints ;
```

Arguments:

userGeometry	Specifies the user-specified geometry or empty string.
borderWidth	Specifies the border width.
defaultGeometry	Specifies the application's default geometry or empty string.
sizeHints	Specifies the size hints for the window in its normal state.
topLeft	Return the x and y offsets.
area	Return the width and height determined.
gravity	Returns the window gravity.

Description:

The **XWMGeometry** function combines any geometry information (given in the format used by XParseGeometry) specified by the user and by the calling program with size hints (usually the ones to be stored in **WM_NORMAL_HINTS**) and returns the position, size, and gravity (**NorthWestGravity**, **NorthEastGravity**, **SouthEastGravity** or **SouthWestGravity**) that describe the window. If the base size is not set in the **XWMSizeHint** list, the minimum size is used if set. Otherwise, a base size of 0 is assumed. If no minimum size is set in the hints list, the base size is used.

Note that invalid geometry specifications can cause a width or height of 0 to be returned.

Chapter 3

Event Reference

3.1 XEvent

Types:

```
datatype 'a XEvent = ButtonPress      of ... | ButtonRelease    of ...
                   | ButtonClick      of ... | CirculateNotify  of ...
                   | CirculateRequest of ... | ColormapNotify   of ...
                   | ConfigureNotify  of ... | ConfigureRequest of ...
                   | CreateNotify     of ... | DestroyNotify    of ...
                   | EnterNotify      of ... | LeaveNotify      of ...
                   | Expose           of ... | FocusIn          of ...
                   | FocusOut         of ... | GraphicsExpose   of ...
                   | NoExpose         of ... | GravityNotify    of ...
                   | KeymapNotify     of ... | KeyPress         of ...
                   | KeyRelease       of ... | MapNotify        of ...
                   | UnmapNotify      of ... | MapRequest       of ...
                   | MotionNotify     of ... | ReparentNotify   of ...
                   | ResizeRequest    of ... | SelectionClear   of ...
                   | SelectionNotify  of ... | SelectionRequest of ...
                   | VisibilityNotify of ... | DeleteRequest    of ...
                   | Message          of ...
```

Description:

The **XEvent** type is a union of the individual types returned for each different type of event. Event handlers typically pattern-match on the **XEvent** members, choosing to match events that they are interested in, and then a default pattern match to provide a default action for all other events. For example:

```
fun Handler (Expose {window,region,...},state) = ...

|   Handler (EnterNotify {window,...},state) = ...
|   Handler (LeaveNotify {window,...},state) = ...

|   Handler (MotionNotify {window,pointer,...},state) = ...

|   Handler (_,state) = state ;  (* default is to do nothing *)
```

All the event types have a `sendEvent` member which is set to true if the event came from a SendEvent protocol request. Most events also contain a `time` member, which is the time at which the event occurred.

3.2 ButtonPress, ButtonRelease, KeyPress, KeyRelease, MotionNotify

Types:

```
datatype Modifier = ShiftMask   | LockMask    | ControlMask
                  | Mod1Mask    | Mod2Mask    | Mod3Mask
                  | Mod4Mask    | Mod5Mask
                  | Button1Mask | Button2Mask | Button3Mask
                  | Button4Mask | Button5Mask
                  | AnyModifier ;

datatype ButtonName = Button1 | Button2 | Button3
                    | Button4 | Button5 | AnyButton ;

ButtonPress   of { sendEvent:   bool,
                   window:      Drawable,
                   root:        Drawable,
                   subwindow:   Drawable,
                   time:        int,
                   pointer:     XPoint,
                   rootPointer: XPoint,
                   modifiers:   Modifier list,
                   button:      ButtonName }

ButtonRelease of { sendEvent:   bool,
                   window:      Drawable,
                   root:        Drawable,
                   subwindow:   Drawable,
                   time:        int,
                   pointer:     XPoint,
                   rootPointer: XPoint,
                   modifiers:   Modifier list,
                   button:      ButtonName }

ButtonClick of { sendEvent:   bool,
                 window:      Drawable,
                 root:        Drawable,
                 subwindow:   Drawable,
                 time:        int,
                 pointer:     XPoint,
                 rootPointer: XPoint,
                 modifiers:   Modifier list,
                 button:      ButtonName,
                 up:          int,          (* number of up   transitions *)
                 down:        int }         (* number of down transitions *)

KeyPress      of { sendEvent:   bool,
                   window:      Drawable,
                   root:        Drawable,
```

```
                    subwindow:   Drawable,
                    time:        int,
                    pointer:     XPoint,
                    rootPointer: XPoint,
                    modifiers:   Modifier list,
                    keycode:     int }

KeyRelease    of { sendEvent:   bool,
                    window:      Drawable,
                    root:        Drawable,
                    subwindow:   Drawable,
                    time:        int,
                    pointer:     XPoint,
                    rootPointer: XPoint,
                    modifiers:   Modifier list,
                    keycode:     int }

MotionNotify  of { sendEvent:   bool,
                    window:      Drawable,
                    root:        Drawable,
                    subwindow:   Drawable,
                    time:        int,
                    pointer:     XPoint,
                    rootPointer: XPoint,
                    modifiers:   Modifier list,
                    isHint:      bool }
```

Description:

These structures have the following common members: window, root, subwindow, time, pointer, rootPointer, and modifiers. The window member is set to the window on which the event was generated and is referred to as the event window. The root member is set to the source window's root window. The rootPointer member is set to the pointer's coordinates relative to the root window's origin at the time of the event.

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 **NoDrawable**. 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 pointer member is set to the coordinates relative to the event window's origin. Otherwise, this member is set to (0,0).

The modifiers member is set to indicate the state of the pointer buttons and modifier keys just prior to the event. It is a list of button or modifier key masks: **Button1Mask**, **Button2Mask**, **Button3Mask**, **Button4Mask**, **Button5Mask**, **ShiftMask**, **LockMask**, **ControlMask**, **Mod1Mask**, **Mod2Mask**, **Mod3Mask**, **Mod4Mask**, **Mod5Mask**.

KeyPress and **KeyRelease** events have a member 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.

ButtonPress and **ButtonRelease** events have a member called button. It represents the pointer button that changed state and can be **Button1**, **Button2**, **Button3**, **Button4**, or **Button5**.

The **ButtonClick** event can be used as an alternative to the **ButtonPress** and **ButtonRelease** combinations. It returns the number of up and down transitions of the pointer

button in a small predetermined time interval. In this way it is easy to detect double and triple clicks.

MotionNotify events have a member called isHint. It can be set to true or false.

3.3 CirculateNotify

Types:

```
datatype Placement = PlaceOnTop | PlaceOnBottom ;

CirculateNotify of { sendEvent: bool,
                     event:     Drawable,
                     window:    Drawable,
                     place:     Placement }
```

Description:

The event member is set either to the restacked window or to its parent, depending on whether **StructureNotifyMask** or **SubstructureNotifyMask** 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.

3.4 CirculateRequest

Types:

```
datatype Placement = PlaceOnTop | PlaceOnBottom ;

CirculateRequest of { sendEvent: bool,
                      parent:    Drawable,
                      window:    Drawable,
                      place:     Placement }
```

Description:

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.

3.5 ColormapNotify

Types:

```
ColormapNotify of { sendEvent: bool,
                    window:    Drawable,
                    colormap:  Colormap,
                    new:       bool,
                    installed: bool }
```

Description:

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 **NoColormap**. 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 installed member is always set to indicate whether the colormap is installed or uninstalled.

3.6 ConfigureNotify

Types:

```
ConfigureNotify of { sendEvent:        bool,
                     event:            Drawable,
                     window:           Drawable,
                     position:         XPoint,
                     size:             XRectangle,
                     borderWidth:      int,
                     above:            Drawable,
                     overrideRedirect: bool }
```

Description:

The event member is set either to the reconfigured window or to its parent, depending on whether **StructureNotifyMask** or **SubstructureNotifyMask** was selected. The window member is set to the window whose size, position, border, and/or stacking order was changed.

The position member is set to the coordinates relative to the parent window's origin and indicates the position of the upper-left outside corner of the window. The size member is set to the inside size of the window, not including the border. The borderWidth 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 **NoDrawable**, 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 overrideRedirect member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the overrideRedirect member is true.

3.7 ConfigureRequest

Types:

```
datatype StackMode = Above | Below | TopIf | BottomIf | Opposite ;

ConfigureRequest of { sendEvent:   bool,
                      parent:      Drawable,
                      window:      Drawable,
                      position:    XPoint,
                      size:        XRectangle,
                      borderWidth: int,
                      above:       Drawable,
                      detail:      StackMode }
```

Description:

The parent member is set to the parent window. The window member is set to the window whose size, position, border width, and/or stacking order is to be reconfigured.

3.8 CreateNotify

Types:

```
CreateNotify of { sendEvent:        bool,
                  parent:           Drawable,
                  window:           Drawable,
                  position:         XPoint,
                  size:             XRectangle,
                  borderWidth:      int,
                  overrideRedirect: bool }
```

Description:

The parent member is set to the created window's parent. The window member specifies the created window. The position member is set to the created window's coordinates relative to the parent window's origin and indicates the position of the upper-left outside corner of the created window. The size member is set to the inside size of the created window (not including the border) and is always nonzero. The borderWidth member is set to the width of the created window's border, in pixels. The overrideRedirect member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the overrideRedirect member is true.

3.9 DeleteRequest

Types:

```
DeleteRequest of { window: Drawable }
```

Description:

This event is generated when the window manager tries to destroy a top level window. Instead of the window being destroyed the window manager sends this event to the application. The application can either ignore the event, or, typically, it can will perform some save operations and then destroy the window itself. The window member is set to the window that is to be destroyed.

3.10 DestroyNotify

Types:

```
DestroyNotify of { sendEvent: bool,
                   event:     Drawable,
                   window:    Drawable }
```

Description:

The event member is set either to the destroyed window or to its parent, depending on whether **StructureNotifyMask** or **SubstructureNotifyMask** was selected. The window member is set to the window that is destroyed.

3.11 EnterNotify, LeaveNotify, NotifyMode, NotifyDetail

Types:

```
datatype NotifyMode = NotifyNormal
                    | NotifyGrab
                    | NotifyUngrab
                    | NotifyWhileGrabbed ;

datatype NotifyDetail = NotifyAncestor         | NotifyVirtual
                      | NotifyInferior         | NotifyNonLinear
                      | NotifyNonLinearVirtual | NotifyPointer
                      | NotifyPointerRoot      | NotifyDetailNone ;

EnterNotify of { sendEvent:   bool,
                 window:      Drawable,
                 root:        Drawable,
                 subwindow:   Drawable,
                 time:        int,
                 pointer:     XPoint,
                 rootPointer: XPoint,
                 mode:        NotifyMode,
                 detail:      NotifyDetail,
                 focus:       bool,
                 modifiers:   Modifier list }

LeaveNotify of { sendEvent:   bool,
                 window:      Drawable,
                 root:        Drawable,
                 subwindow:   Drawable,
```

```
time:        int,
pointer:     XPoint,
rootPointer: XPoint,
mode:        NotifyMode,
detail:      NotifyDetail,
focus:       bool,
modifiers:   Modifier list }
```

Description:

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 **NoDrawable**. 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 **NoDrawable**.

The time member is set to the time when the event was generated and is expressed in milliseconds. The pointer member is 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, pointer is the pointer coordinates relative to the event window's origin. Otherwise, pointer is set to (0,0). The rootPointer member is set to the pointer's coordinates relative to the root window's origin at the time of the event.

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 modifiers member is set to indicate the state of the pointer buttons and modifier keys just prior to the event. It is a list of 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**.

3.12 Expose

Types:

```
Expose of { sendEvent: bool,
            window:    Drawable,
            region:    XRectangle,
            count:     int }
```

Description:

The window member is set to the exposed (damaged) window. The region member is set to the damaged area within the window. 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.

3.13 FocusIn, FocusOut

Types:

```
FocusIn of { sendEvent: bool,
             window:    Drawable,
             mode:      NotifyMode,
             detail:    NotifyDetail }

FocusOut of { sendEvent: bool,
              window:    Drawable,
              mode:      NotifyMode,
              detail:    NotifyDetail }
```

Description:

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**.

3.14 GraphicsExpose, NoExpose

Types:

```
datatype GraphicsCode = CopyArea | CopyPlane ;

GraphicsExpose of { sendEvent: bool,
                    window:    Drawable,
                    region:    XRectangle,
                    count:     int,
                    code:      GraphicsCode }
```

```
NoExpose of { sendEvent: bool,
              window:    Drawable,
              code:      GraphicsCode }
```

Description:

Both structures have drawable and code as common members. The drawable member is set to the drawable of the destination region on which the graphics request was to be performed. The code member is set to the graphics request initiated by the client and can be either **CopyArea** or **CopyPlane**. If it is **CopyArea**, a call to **XCopyArea** initiated the request. If it is **CopyPlane**, a call to **XCopyPlane** initiated the request.

The **GraphicsExpose** structure has these additional members: region, and count. The region member is set to the area within the drawable. 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.

3.15 GravityNotify

Types:

```
GravityNotify of { sendEvent: bool,
                   event:     Drawable,
                   window:    Drawable,
                   position:  XPoint }
```

Description:

The event member is set either to the window that was moved or to its parent, depending on whether **StructureNotifyMask** or **SubstructureNotifyMask** was selected. The window member is set to the child window that was moved. The position member is set to the coordinates relative to the new parent window's origin and indicates the position of the upper-left outside corner of the window.

3.16 KeymapNotify

Types:

```
KeymapNotify of { sendEvent: bool,
                  window:    Drawable,
                  keyVector: bool list (* 256 bools *) }
```

Description:

The keyVector member is set to the bit vector of the keyboard. The vector is returned as a list of 256 bools, representing the keys 0 to 255 in that order. Each bool set to true indicates that the corresponding key is currently pressed.

3.17 MapNotify

Types:

```
MapNotify of { sendEvent:        bool,
               event:            Drawable,
               window:           Drawable,
               overrideRedirect: bool }
```

Description:

The event member is set either to the window that was mapped or to its parent, depending on whether **StructureNotifyMask** or **SubstructureNotifyMask** was selected. The window member is set to the window that was mapped. The overrideRedirect 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.

3.18 MapRequest

Types:

```
MapRequest of { sendEvent: bool,
                parent:    Drawable,
                window:    Drawable }
```

Description:

The parent member is set to the parent window. The window member is set to the window to be mapped.

3.19 Message

Types:

```
Message of { window: Drawable, message: 'a } ;
```

Description:

This event is received when a message is sent to a window. The only way to send a message to a window is to call the message-sender function returned by **XSetHandler**. This will only allow a strongly typed messages to be sent.

3.20 ReparentNotify

Types:

```
ReparentNotify of { sendEvent:        bool,
                    event:            Drawable,
                    window:           Drawable,
                    parent:           Drawable,
                    position:         XPoint,
                    overrideRedirect: bool }
```

Description:

The event member is set either to the reparented window or to the old or the new parent, depending on whether **StructureNotifyMask** or **SubstructureNotifyMask** was selected. The window member is set to the window that was reparented. The parent member is set to the new parent window. The position member is set to the reparented window's coordinates relative to the new parent window's origin and defines the upper-left outer corner of the reparented window. The overrideRedirect 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 overrideRedirect member is true.

3.21 ResizeRequest

Types:

```
ResizeRequest of { sendEvent: bool,
                   window:    Drawable,
                   size:      XRectangle }
```

Description:

The window member is set to the window whose size another client attempted to change. The size member is set to the inside size of the window, excluding the border.

3.22 SelectionClear

Types:

```
SelectionClear of { sendEvent: bool,
                    window:    Drawable,
                    selection: int,
                    time:      int }
```

Description:

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.

3.23 SelectionNotify

Types:

```
SelectionNotify of { sendEvent: bool,
                     requestor: Drawable,
                     selection: int,
                     target:    int,
                     property:  int,
                     time:      int }
```

Description:

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, **XA_PRIMARY** is used for the primary selection. The target member is set to the atom that indicates the converted type. For example, **XA_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 zero. The time member is set to the time the conversion took place and can be a timestamp or **CurrentTime**.

3.24 SelectionRequest

Types:

```
SelectionRequest of { sendEvent: bool,
                      owner:     Drawable,
                      requestor: Drawable,
                      selection: int,
                      target:    int,
                      property:  int,
                      time:      int }
```

Description:

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, **XA_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 zero. The time member is set to the time and is a timestamp or **CurrentTime** from the **XConvertSelection** request.

3.25 UnmapNotify

Types:

```
UnmapNotify of { sendEvent:     bool,
                 event:         Drawable,
                 window:        Drawable,
                 fromConfigure: bool }
```

Description:

The event member is set either to the unmapped window or to its parent, depending on whether **StructureNotifyMask** or **SubstructureNotifyMask** 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 fromConfigure 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 winGravity of **UnmapGravity**.

3.26 VisibilityNotify

Types:

```
datatype Visibility = VisibilityUnobscured
                    | VisibilityPartiallyObscured
                    | VisibilityObscured ;

VisibilityNotify of { sendEvent:  bool,
                      window:     Drawable,
                      visibility: Visibility }
```

Description:

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 **VisibilityObscured**. 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 **VisibilityNotify** 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 **VisibilityNotify** 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 **VisibilityNotify** structure set to **VisibilityObscured**.

Chapter 4

Protocol Error Messages

4.1 BadAccess

Description:

XFreeColors	pixel not allocated.
XSelectInput	selecting event that can only be selected by one client at a time, when another client already has it selected.
XStoreColors	pixel not allocated, or allocated read-only.
XStoreNamedColor	pixel not allocated, or allocated read-only.

4.2 BadAlloc

Description:

The server failed to allocate the requested resource.

4.3 BadAtom

Description:

A value for an atom argument does not name a defined atom.

4.4 BadColor

Description:

A value for a **Colormap** argument does not name a defined **Colormap**. ML type-checking should avoid this error.

4.5 BadCursor

Description:

A value for a **Cursor** argument does not name a defined **Cursor**. ML type-checking should avoid this error.

4.6 BadDrawable

Description:

A value for a **Drawable** argument does not name a defined Window or Pixmap. ML type-checking should avoid this error.

4.7 BadFont

Description:

A value for a **Font** argument does not name a defined **Font**. ML type-checking should avoid this error.

4.8 BadGC

Description:

A value for a **GC** argument does not name a defined **GC**. ML type-checking should avoid this error.

4.9 BadImplementation

Description:

The server does not implement some aspect of the request. This should never occur in Xlib since only standard requests are made.

4.10 BadIDChoice, BadLength

Description:

Internal Xlib error.

4.11 BadMatch

Description:

Some argument, or arguments, have the correct type and range, but fail to 'match' in some other way.

XChangeWindowAtrributes	Changing the background or border of an **InputOnlyClass** window.
XClearArea	**InputOnlyClass** window specified.
XClearWindow	**InputOnlyClass** window specified.
XConfigureWindow	Sibling incorrectly specified, or changing the background or border of an **InputOnlyClass** window.
XCopyArea	The drawables must have the same root and depth.
XCopyPlane	The drawables must have the same root.
XCreateColormap	**Visual** type not supported, or bad **AllocType**.
XSetWindowColormap	**Colormap** has different visual type to window.
XCreatePixmapCursor	Masks must have depth of 1, and must be the same size, and the hotspot must be within that size.
XChangeGC	Tile pixmap must have the same root and depth as the **GC**. Stipple pixmap must have depth of 1 and must have the same root as the **GC**. Clip-mask pixmap must have depth of 1 and must have the same root as the **GC**. Using a **GC** with a **Drawable** of different root or depth results in **BadMatch**.
XPutImage	If **XYBitmap** format is used, the depth must be 1. For **XYPixmap** and **ZPixmap**, the depth must match the depth of the drawable.
XGetImage	Specified area not within the source drawable.
XGetSubImage	Specified area not within the source drawable.
XRestackWindows	Specified window is not child window.
XCreatePixmapFromBitmapData	Depth must be supported by screen.
XReparentWindow	New parent is not on same screen as old parent.
XSetClipRectangles	Incorrect ordering.
XSetInputFocus	Focus window must be viewable.

4.12 BadPixmap

Description:

A value for a Pixmap does not name a defined Pixmap. ML type-checking should avoid this error.

4.13 BadRequest

Description:

This should never occur in Xlib since only standard requests are made.

4.14 BadValue

Description:

Some numeric value falls outside the range of values accepted by the request.

XAllocColorCells	Number of colours must be positive and planes must be non-negative.
XAllocColorPlanes	Number of colours must be positive, and reds, green and blues must be non-negative
XFreeColors	Specified pixel is not a valid index into the colormap.
XBell	Percent must be ˜100 to 100.
XResizeWindow	Window width must be non-zero.
XCopyPlane	Plane must have one bit set to 1, and specify an existing plane.
XCreateGlyphCursor	Source char and mask char must exist in the font.
XSetDashes	Dash elements must be positive and less than 256.
XCreatePixmap	Specified width must be non-zero, and depth must be supported.
XSetScreenSaver	Incorrect timeout value.
XStoreColors	Specified pixel is not a valid index into the colormap.

4.15 BadWindow

Description:

A value for a Window does not name a defined Window. ML type-checking should avoid this error.

Index

D

E

F

G

H

I

R

S

T

U

V

W

X

Y

Z