

Introduction

This section contains the documentation for the INTERIX X11R5 Distribution.



Chapter 1

X Library

The routines in this section comprise the X library.

To use these routines, include the header files in **</usr/X11/include/X11/X.h>**.

NAME

XAllocClassHint(), XSetClassHint(), XGetClassHint(), XClassHint() – allocate class hints structure and set or read a window's WM_CLASS property

SYNOPSIS

```
XClassHint *XAllocClassHint (void)
XSetClassHint (Display *display, Window w, XClassHint *class_hints)
Status XGetClassHint (Display *display, Window w, XClassHint *class_hints_return)
```

ARGUMENTS

display Specifies the connection to the X server.
class_hints Specifies the **XClassHint** structure that is to be used.
class_hints_return Returns the **XClassHint** structure.
w Specifies the window.

DESCRIPTION

The *XAllocClassHint()* function allocates and returns a pointer to a **XClassHint** structure. Note that the pointer fields in the **XClassHint** structure are initially set to NULL. If insufficient memory is available, *XAllocClassHint()* returns NULL. To free the memory allocated to this structure, use *XFree()*.

The *XSetClassHint()* function sets the class hint for the specified window. If the strings are not in the Host Portable Character Encoding the result is implementation dependent.

XSetClassHint() can generate BadAlloc and BadWindow errors.

The *XGetClassHint()* function returns the class hint of the specified window to the members of the supplied structure. If the data returned by the server is in the Latin Portable Character Encoding, then the returned strings are in the Host Portable Character Encoding. Otherwise, the result is implementation dependent. It returns nonzero status on success; otherwise it returns a zero status. To free *res_name* and *res_class* when finished with the strings, use *XFree()* on each individually.

XGetClassHint() can generate a BadWindow error.

PROPERTIES

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

STRUCTURES

The **XClassHint** structure contains:

```
typedef struct {
    char *res_name;
    char *res_class;
} XClassHint;
```

The *res_name* member contains the application name, and the *res_class* member contains the application class. Note that the name set in this property may differ from the name set as WM_NAME. That is, WM_NAME specifies what should be displayed in the title bar and, therefore, can contain temporal information (for example, the name of a file

currently in an editor's buffer). On the other hand, the name specified as part of WM_CLASS is the formal name of the application that should be used when retrieving the application's resources from the resource database.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocIconSize(), *XAllocSizeHints()*, *XAllocWMHints()*, *XFree()*, *XSetCommand()*, *XSetTransientForHint()*, *XSetTextProperty()*, *XSetWMClientMachine()*, *XSetWMColormapWindows()*, *XSetWMIconName()*, *XSetWMName()*, *XSetWMProperties()*, *XSetWMProtocols()*, *XStringListToTextProperty()*

Xlib — C Language X Interface

NAME

XAllocIconSize(), XSetIconSizes(), XGetIconSizes(), XIconSize() – allocate icon size structure and set or read a window's WM_ICON_SIZES property

SYNOPSIS

```
XIconSize *XAllocIconSize ( )
XSetIconSizes (Display *display, Window w,
                XIconSize *size_list, int count)
Status XGetIconSizes (Display *display, Window w,
                     XIconSize **size_list_return, int *count_return)
```

ARGUMENTS

display Specifies the connection to the X server.
count Specifies the number of items in the size list.
count_return
 Returns the number of items in the size list.
size_list Specifies the size list.
size_list_return
 Returns the size list.
w Specifies the window.

DESCRIPTION

The *XAllocIconSize()* function allocates and returns a pointer to a **XIconSize** structure. Note that all fields in the **XIconSize** structure are initially set to zero. If insufficient memory is available, *XAllocIconSize()* returns NULL. To free the memory allocated to this structure, use *XFree()*.

The *XSetIconSizes()* function is used only by window managers to set the supported icon sizes.

XSetIconSizes() can generate BadAlloc and BadWindow errors.

The *XGetIconSizes()* function returns zero if a window manager has not set icon sizes; otherwise, it return nonzero. *XGetIconSizes()* should be called by an application that wants to find out what icon sizes would be most appreciated by the window manager under which the application is running. The application should then use *XSetWMHints()* to supply the window manager with an icon pixmap or window in one of the supported sizes. To free the data allocated in *size_list_return*, use *XFree()*.

XGetIconSizes() can generate a BadWindow error.

PROPERTIES

WM_ICON_SIZES
 The window manager may set this property on the root window to specify the icon sizes it supports. The C type of this property is **XIconSize**.

STRUCTURES

The **XIconSize** structure contains:

```
typedef struct {
    int min_width, min_height;
    int max_width, max_height;
    int width_inc, height_inc;
} XIconSize;
```

The *width_inc* and *height_inc* members define an arithmetic progression of sizes (minimum to maximum) that represent the supported icon sizes.

DIAGNOSTICS**BadAlloc**

The server failed to allocate the requested resource or server memory.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), *XAllocSizeHints()*, *XAllocWMHints()*, *XFree()*, *XSetCommand()*, *XSetTransientForHint()*, *XSetTextProperty()*, *XSetWMClientMachine()*, *XSetWMColormapWindows()*, *XSetWMIconName()*, *XSetWMName()*, *XSetWMProperties()*, *XSetWMProtocols()*, *XStringListToTextProperty()*

Xlib — C Language X Interface

NAME

XAllocStandardColormap(), XSetRGBColormaps(), XGetRGBColormaps(),
XStandardColormap() – allocate, set, or read a standard colormap structure

SYNOPSIS

```
XStandardColormap *XAllocStandardColormap ( )
void XSetRGBColormaps (Display *display, Window w,
                       XStandardColormap *std_colormap, int count,
                       Atom property)
Status XGetRGBColormaps (Display *display, Window w,
                        XStandardColormap **std_colormap_return,
                        int *count_return, Atom property)
```

ARGUMENTS

display Specifies the connection to the X server.
count Specifies the number of colormaps.
count_return
 Returns the number of colormaps.
property Specifies the property name.
std_colormap
 Specifies the **XStandardColormap** structure to be used.
std_colormap_return
 Returns the **XStandardColormap** structure.

DESCRIPTION

The *XAllocStandardColormap()* function allocates and returns a pointer to a **XStandardColormap** structure. Note that all fields in the **XStandardColormap** structure are initially set to zero. If insufficient memory is available, *XAllocStandardColormap()* returns NULL. To free the memory allocated to this structure, use *XFree()*.

The *XSetRGBColormaps()* function replaces the RGB colormap definition in the specified property on the named window. If the property does not already exist, *XSetRGBColormaps()* sets the RGB colormap definition in the specified property on the named window. The property is stored with a type of RGB_COLOR_MAP and a format of 32. Note that it is the caller's responsibility to honor the ICCCM restriction that only RGB_DEFAULT_MAP contain more than one definition.

The *XSetRGBColormaps()* function usually is only used by window or session managers. To create a standard colormap, follow this procedure:

1. Open a new connection to the same server.
2. Grab the server.
3. See if the property is on the property list of the root window for the screen.
4. If the desired property is not present:
 - Create a colormap (unless using the default colormap of the screen).
 - Determine the color characteristics of the visual.
 - Call *XAllocColorPlanes()* or *XAllocColorCells()* to allocate cells in the colormap.
 - Call *XStoreColors()* to store appropriate color values in the colormap.

- Fill in the descriptive members in the **XStandardColormap** structure.
- Attach the property to the root window.
- Use *XSetCloseDownMode()* to make the resource permanent.

5. Ungrab the server.

XSetRGBColormaps() can generate BadAlloc, BadAtom, and BadWindow errors.

The *XGetRGBColormaps()* function returns the RGB colormap definitions stored in the specified property on the named window. If the property exists, is of type RGB_COLOR_MAP, is of format 32, and is long enough to contain a colormap definition, *XGetRGBColormaps()* allocates and fills in space for the returned colormaps and returns a nonzero status. If the visualid is not present, *XGetRGBColormaps()* assumes the default visual for the screen on which the window is located; if the killid is not present, *None()* is assumed, which indicates that the resources cannot be released. Otherwise, none of the fields are set, and *XGetRGBColormaps()* returns a zero status. Note that it is the caller's responsibility to honor the ICCCM restriction that only RGB_DEFAULT_MAP contain more than one definition.

XGetRGBColormaps() can generate BadAtom and BadWindow errors.

STRUCTURES

The **XStandardColormap** structure contains:

Hints

```
#define ReleaseByFreeingColormap    ( (XID) 1L)
```

Values

```
typedef struct {
    Colormap colormap;
    unsigned long red_max;
    unsigned long red_mult;
    unsigned long green_max;
    unsigned long green_mult;
    unsigned long blue_max;
    unsigned long blue_mult;
    unsigned long base_pixel;
    VisualID visualid;
    XID killid;
} XStandardColormap;
```

The *colormap* member is the colormap created by the *XCreateColormap()* function. The *red_max*, *green_max*, and *blue_max* members give the maximum red, green, and blue values, respectively. Each color coefficient ranges from zero to its max, inclusive. For example, a common colormap allocation is 3/3/2 (3 planes for red, 3 planes for green, and 2 planes for blue). This colormap would have *red_max* = 7, *green_max* = 7, and *blue_max* = 3. An alternate allocation that uses only 216 colors is *red_max* = 5, *green_max* = 5, and *blue_max* = 5.

The *red_mult*, *green_mult*, and *blue_mult* members give the scale factors used to compose a full pixel value. (See the discussion of the *base_pixel* members for further information.) For a 3/3/2 allocation, *red_mult* might be 32, *green_mult* might be 4, and *blue_mult* might be 1. For a 6-colors-each allocation, *red_mult* might be 36, *green_mult* might be 6, and

blue_mult might be 1.

The *base_pixel* member gives the base pixel value used to compose a full pixel value. Usually, the *base_pixel* is obtained from a call to the *XAllocColorPlanes()* function. Given integer red, green, and blue coefficients in their appropriate ranges, one then can compute a corresponding pixel value by using the following expression:

$$(r * red_mult + g * green_mult + b * blue_mult + base_pixel) \& 0xFFFFFFFF$$

For *GrayScale()* colormaps, only the *colormap*, *red_max*, *red_mult*, and *base_pixel* members are defined. The other members are ignored. To compute a GrayScale pixel value, use the following expression:

$$(gray * red_mult + base_pixel) \& 0xFFFFFFFF$$

Negative multipliers can be represented by converting the 2's complement representation of the multiplier into an unsigned long and storing the result in the appropriate *_mult* field. The step of masking by 0xFFFFFFFF effectively converts the resulting positive multiplier into a negative one. The masking step will take place automatically on many machine architectures, depending on the size of the integer type used to do the computation,

The *visualid* member gives the ID number of the visual from which the colormap was created. The *killid* member gives a resource ID that indicates whether the cells held by this standard colormap are to be released by freeing the colormap ID or by calling the *XKillClient()* function on the indicated resource. (Note that this method is necessary for allocating out of an existing colormap.)

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

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadAtom

A value for an Atom argument does not name a defined Atom.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocColor(), *XCreateColormap()*, *XFree()*, *XSetCloseDownMode()*

Xlib — C Language X Interface

NAME

XButtonEvent, XKeyEvent, XMotionEvent – KeyPress, KeyRelease, ButtonPress, ButtonRelease, and MotionNotify event structures

STRUCTURES

The structures for KeyPress, KeyRelease, ButtonPress, ButtonRelease, and MotionNotify events contain:

```
typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Window window;
    Window root;
    Window subwindow;
    Time time;
    int x, y;
    int x_root, y_root;
    unsigned int state;
    unsigned int button;
    Bool same_screen;
} XButtonEvent;
```

	ButtonPress or ButtonRelease
	# of last request processed by server
	true if this came from a SendEvent request
	Display the event was read from
	“event” window it is reported relative to
	root window that the event occurred on
	child window
	milliseconds
	pointer x, y coordinates in event window
	coordinates relative to root
	key or button mask
	detail
	same screen flag

```
typedef XButtonEvent XButtonPressedEvent;
typedef XButtonEvent XButtonReleasedEvent;
```

```
typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Window window;
    Window root;
    Window subwindow;
    Time time;
    int x, y;
    int x_root, y_root;
    unsigned int state;
    unsigned int keycode;
    Bool same_screen;
} XKeyEvent;
```

	KeyPress or KeyRelease
	# of last request processed by server
	true if this came from a SendEvent request
	Display the event was read from
	“event” window it is reported relative to
	root window that the event occurred on
	child window
	milliseconds
	pointer x, y coordinates in event window
	coordinates relative to root
	key or button mask
	detail
	same screen flag

```
typedef XKeyEvent XKeyPressedEvent;
typedef XKeyEvent XKeyReleasedEvent;
```

```
typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Window window;
```

	MotionNotify
	# of last request processed by server
	true if this came from a SendEvent request
	Display the event was read from
	“event” window reported relative to

Window root;	root window that the event occurred on
Window subwindow;	child window
Time time;	milliseconds
int x, y;	pointer x, y coordinates in event window
int x_root, y_root;	coordinates relative to root
unsigned int state;	key or button mask
char is_hint;	detail
Bool same_screen;	same screen flag

```
} XMotionEvent;
```

```
typedef XMotionEvent XPointerMovedEvent;
```

When you receive these events, their structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

These structures have the following common members: *window*, *root*, *subwindow*, *time*, *x*, *y*, *x_root*, *y_root*, *state*, and *same_screen*. The *window* member is set to the window on which the event was generated and is referred to as the event window. As long as the conditions previously discussed are met, this is the window used by the X server to report the event. The *root* member is set to the source window's root window. The *x_root* and *y_root* members are set to the pointer's coordinates relative to the root window's origin at the time of the event.

The *same_screen* member is set to indicate whether the event window is on the same screen as the root window and can be either True or False. If True, the event and root windows are on the same screen. If False, the event and root windows are not on the same screen.

If the source window is an inferior of the event window, the *subwindow* member of the structure is set to the child of the event window that is the source window or the child of the event window that is an ancestor of the source window. Otherwise, the X server sets the *subwindow* member to None. The *time* member is set to the time when the event was generated and is expressed in milliseconds.

If the *event* window is on the same screen as the root window, the *x* and *y* members are set to the coordinates relative to the event window's origin. Otherwise, these members are set to zero.

The *state* member is set to indicate the logical state of the pointer buttons and modifier keys just prior to the event, which is the bitwise inclusive OR of one or more of the button or modifier key masks: Button1Mask, Button2Mask, Button3Mask, Button4Mask, Button5Mask, ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, and Mod5Mask.

Each of these structures also has a member that indicates the detail. For the **XKeyPressedEvent** and **XKeyReleasedEvent** structures, this member is called keycode. It is set to a number that represents a physical key on the keyboard. The keycode is an

arbitrary representation for any key on the keyboard (see sections 12.7 and 16.1).

For the **XButtonPressedEvent** and **XButtonReleasedEvent** structures, this member is called `button`. It represents the pointer button that changed state and can be the `Button1`, `Button2`, `Button3`, `Button4`, or `Button5` value. For the **XPointerMovedEvent** structure, this member is called `is_hint`. It can be set to `NotifyNormal` or `NotifyHint`.

SEE ALSO

XAnyEvent(), *XCreateWindowEvent()*, *XCirculateEvent()*, *XCirculateRequestEvent()*, *XColormapEvent()*, *XConfigureEvent()*, *XConfigureRequestEvent()*, *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*, *XExposeEvent()*, *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*, *XKeymapEvent()*, *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*, *XResizeRequestEvent()*, *XSelectionClearEvent()*, *XSelectionEvent()*, *XSelectionRequestEvent()*, *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XChangeKeyboardControl(), XGetKeyboardControl(), XAutoRepeatOn(), XAutoRepeatOff(), XBell(), XQueryKeymap(), XKeyboardControl() – manipulate keyboard settings and keyboard control structure

SYNOPSIS

```
XChangeKeyboardControl (Display *display, unsigned long value_mask,
                        XKeyboardControl *values)
XGetKeyboardControl (Display *display, XKeyboardState *values_return)
XAutoRepeatOn (Display *display)
XAutoRepeatOff (Display *display)
XBell (Display *display, int percent)
XQueryKeymap (Display *display, char keys_return[32])
```

ARGUMENTS

display Specifies the connection to the X server.

keys_return Returns an array of bytes that identifies which keys are pressed down. Each bit represents one key of the keyboard.

percent Specifies the volume for the bell, which can range from -100 to 100 inclusive.

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

values Specifies one value for each bit set to 1 in the mask.

values_return Returns the current keyboard controls in the specified **XKeyboardState** structure.

DESCRIPTION

The *XChangeKeyboardControl()* function controls the keyboard characteristics defined by the **XKeyboardControl** structure. The *value_mask* argument specifies which values are to be changed.

XChangeKeyboardControl() can generate BadMatch and BadValue errors.

The *XGetKeyboardControl()* function returns the current control values for the keyboard to the **XKeyboardState** structure.

The *XAutoRepeatOn()* function turns on auto-repeat for the keyboard on the specified display.

The *XAutoRepeatOff()* function turns off auto-repeat for the keyboard on the specified display.

The *XBell()* function rings the bell on the keyboard on the specified display, if possible. The specified volume is relative to the base volume for the keyboard. If the value for the percent argument is not in the range -100 to 100 inclusive, a BadValue error results. The volume at which the bell rings when the percent argument is nonnegative is:

$$\text{base} - [(\text{base} * \text{percent}) / 100] + \text{percent}$$

The volume at which the bell rings when the percent argument is negative is:

$$\text{base} + [(\text{base} * \text{percent}) / 100]$$

To change the base volume of the bell, use *XChangeKeyboardControl()*.

XBell() can generate a *BadValue* error.

The *XQueryKeymap()* function returns a bit vector for the logical state of the keyboard, where each bit set to 1 indicates that the corresponding key is currently pressed down. The vector is represented as 32 bytes. Byte *N* (from 0) contains the bits for keys *8N* to *8N + 7* with the least-significant bit in the byte representing key *8N*.

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

STRUCTURES

The **XKeyboardControl** structure contains:

Mask bits for *ChangeKeyboardControl*

```
#define KBKeyClickPercent    (1L<<0)
#define KBBellPercent        (1L<<1)
#define KBBellPitch          (1L<<2)
#define KBBellDuration       (1L<<3)
#define KBLed                (1L<<4)
#define KBLedMode            (1L<<5)
#define KBKey                (1L<<6)
#define KBAutoRepeatMode     (1L<<7)
```

Values

```
typedef struct {
    int key_click_percent;
    int bell_percent;
    int bell_pitch;
    int bell_duration;
    int led;
    int led_mode;           LedModeOn, LedModeOff
    int key;
    int auto_repeat_mode;   AutoRepeatModeOff, AutoRepeatModeOn,
                           AutoRepeatModeDefault
} XKeyboardControl;
```

The *key_click_percent* member sets the volume for key clicks between 0 (off) and 100 (loud) inclusive, if possible. A setting of -1 restores the default. Other negative values generate a *BadValue* error.

The *bell_percent* sets the base volume for the bell between 0 (off) and 100 (loud) inclusive, if possible. A setting of -1 restores the default. Other negative values generate a *BadValue* error. The *bell_pitch* member sets the pitch (specified in Hz) of the bell, if possible. A setting of -1 restores the default. Other negative values generate a *BadValue* error. The *bell_duration* member sets the duration of the bell specified in milliseconds, if possible. A setting of -1 restores the default. Other negative values generate a *BadValue* error.

If both the *led_mode* and *led* members are specified, the state of that LED is changed, if possible. The *led_mode* member can be set to *LedModeOn* or *LedModeOff*. If only *led_mode* is specified, the state of all LEDs are changed, if possible. At most 32 LEDs numbered from one are supported. No standard interpretation of LEDs is defined. If *led*

is specified without *led_mode*, a BadMatch error results.

If both the *auto_repeat_mode* and *key* members are specified, the *auto_repeat_mode* of that key is changed (according to `AutoRepeatModeOn`, `AutoRepeatModeOff`, or `AutoRepeatModeDefault`), if possible. If only *auto_repeat_mode* is specified, the global *auto_repeat_mode* for the entire keyboard is changed, if possible, and does not affect the per key settings. If a key is specified without an *auto_repeat_mode*, a BadMatch error results. Each key has an individual mode of whether or not it should auto-repeat and a default setting for the mode. In addition, there is a global mode of whether auto-repeat should be enabled or not and a default setting for that mode. When global mode is `AutoRepeatModeOn`, keys should obey their individual auto-repeat modes. When global mode is `AutoRepeatModeOff`, no keys should auto-repeat. An auto-repeating key generates alternating `KeyPress` and `KeyRelease` events. When a key is used as a modifier, it is desirable for the key not to auto-repeat, regardless of its auto-repeat setting.

The **XKeyboardState** structure contains:

```
typedef struct {
    int key_click_percent;
    int bell_percent;
    unsigned int bell_pitch, bell_duration;
    unsigned long led_mask;
    int global_auto_repeat;
    char auto_repeats[32];
} XKeyboardState;
```

For the LEDs, the least-significant bit of *led_mask* corresponds to LED one, and each bit set to 1 in *led_mask* indicates an LED that is lit. The *global_auto_repeat* member can be set to `AutoRepeatModeOn` or `AutoRepeatModeOff`. The *auto_repeats* member is a bit vector. Each bit set to 1 indicates that auto-repeat is enabled for the corresponding key. The vector is represented as 32 bytes. Byte *N* (from 0) contains the bits for keys 8*N* to 8*N* + 7 with the least-significant bit in the byte representing key 8*N*.

DIAGNOSTICS

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XChangeKeyboardMapping(), *XSetPointerMapping()*

Xlib — C Language X Interface

NAME

XChangeKeyboardMapping(), XGetKeyboardMapping(), XDisplayKeycodes(), XSetModifierMapping(), XGetModifierMapping(), XNewModifiermap(), XInsertModifiermapEntry(), XDeleteModifiermapEntry(), XFreeModifierMap(), XModifierKeymap() – manipulate keyboard encoding and keyboard encoding structure

SYNOPSIS

```
XChangeKeyboardMapping (Display *display, int first_keycode,
                        int keysyms_per_keycode, KeySym *keysyms,
                        int num_codes)
KeySym *XGetKeyboardMapping (Display *display, KeyCode first_keycode,
                             int keycode_count,
                             int *keysyms_per_keycode_return)
XDisplayKeycodes (Display *display, int *min_keycodes_return,
                  int *max_keycodes_return)
int XSetModifierMapping (Display *display, XModifierKeymap *modmap)
XModifierKeymap *XGetModifierMapping (Display *display)
XModifierKeymap *XNewModifiermap (int max_keys_per_mod)
XModifierKeymap *XInsertModifiermapEntry (XModifierKeymap *modmap,
                                          KeyCode keycode_entry,
                                          int modifier)
XModifierKeymap *XDeleteModifiermapEntry (XModifierKeymap *modmap,
                                          KeyCode keycode_entry,
                                          int modifier)
XFreeModifiermap (XModifierKeymap *modmap)
```

ARGUMENTS

display Specifies the connection to the X server.

first_keycode Specifies the first KeyCode that is to be changed or returned.

keycode_count Specifies the number of KeyCodes that are to be returned.

keycode_entry Specifies the KeyCode.

keysyms Specifies an array of KeySyms.

keysyms_per_keycode Specifies the number of KeySyms per KeyCode.

keysyms_per_keycode_return Returns the number of KeySyms per KeyCode.

max_keys_per_mod Specifies the number of KeyCode entries preallocated to the modifiers in the map.

max_keycodes_return Returns the maximum number of KeyCodes.

min_keycodes_return Returns the minimum number of KeyCodes.

modifier Specifies the modifier.

modmap Specifies the **XModifierKeymap** structure.

num_codes Specifies the number of KeyCodes that are to be changed.

DESCRIPTION

The *XChangeKeyboardMapping()* function defines the symbols for the specified number of KeyCodes starting with *first_keycode*. The symbols for KeyCodes outside this range remain unchanged. The number of elements in *keysyms* must be:

$$\text{num_codes} * \text{keysyms_per_keycode}$$

The specified *first_keycode* must be greater than or equal to *min_keycode* returned by *XDisplayKeycodes()*, or a BadValue error results. In addition, the following expression must be less than or equal to *max_keycode* as returned by *XDisplayKeycodes()*, or a BadValue error results:

$$\text{first_keycode} + \text{num_codes} - 1$$

KeySym number N, counting from zero, for KeyCode K has the following index in *keysyms*, counting from zero:

$$(\text{K} - \text{first_keycode}) * \text{keysyms_per_keycode} + \text{N}$$

The specified *keysyms_per_keycode* can be chosen arbitrarily by the client to be large enough to hold all desired symbols. A special KeySym value of NoSymbol should be used to fill in unused elements for individual KeyCodes. It is legal for NoSymbol to appear in nontrailing positions of the effective list for a KeyCode.

XChangeKeyboardMapping() generates a MappingNotify event.

There is no requirement that the X server interpret this mapping. It is merely stored for reading and writing by clients.

XChangeKeyboardMapping() can generate BadAlloc and BadValue errors.

The *XGetKeyboardMapping()* function returns the symbols for the specified number of KeyCodes starting with *first_keycode*. The value specified in *first_keycode* must be greater than or equal to *min_keycode* as returned by *XDisplayKeycodes()*, or a BadValue error results. In addition, the following expression must be less than or equal to *max_keycode* as returned by *XDisplayKeycodes()*:

$$\text{first_keycode} + \text{keycode_count} - 1$$

If this is not the case, a BadValue error results. The number of elements in the KeySyms list is:

$$\text{keycode_count} * \text{keysyms_per_keycode_return}$$

KeySym number N, counting from zero, for KeyCode K has the following index in the list, counting from zero:

$$(\text{K} - \text{first_code}) * \text{keysyms_per_code_return} + \text{N}$$

The X server arbitrarily chooses the *keysyms_per_keycode_return* value to be large enough to report all requested symbols. A special KeySym value of NoSymbol is used to fill in unused elements for individual KeyCodes. To free the storage returned by *XGetKeyboardMapping()*, use *XFree()*.

XGetKeyboardMapping() can generate a BadValue error.

The *XDisplayKeycodes()* function returns the min-keycodes and max-keycodes supported by the specified display. The minimum number of KeyCodes returned is never less than 8, and the maximum number of KeyCodes returned is never greater than 255. Not all KeyCodes in this range are required to have corresponding keys.

The *XSetModifierMapping()* function specifies the KeyCodes of the keys (if any) that are to be used as modifiers. If it succeeds, the X server generates a *MappingNotify()* event, and *XSetModifierMapping()* returns MappingSuccess. X permits at most eight modifier keys. If more than eight are specified in the **XModifierKeymap** structure, a BadLength error results.

The *modifiermap* member of the **XModifierKeymap** structure contains eight sets of *max_keypermod* KeyCodes, one for each modifier in the order Shift, Lock, Control, Mod1, Mod2, Mod3, Mod4, and Mod5. Only nonzero KeyCodes have meaning in each set, and zero KeyCodes are ignored. In addition, all of the nonzero KeyCodes must be in the range specified by *min_keycode* and *max_keycode* in the **Display** structure, or a BadValue error results.

An X server can impose restrictions on how modifiers can be changed, for example, if certain keys do not generate up transitions in hardware, if auto-repeat cannot be disabled on certain keys, or if multiple modifier keys are not supported. If some such restriction is violated, the status reply is MappingFailed, and none of the modifiers are changed. If the new KeyCodes specified for a modifier differ from those currently defined and any (current or new) keys for that modifier are in the logically down state, *XSetModifierMapping()* returns MappingBusy, and none of the modifiers is changed.

XSetModifierMapping() can generate BadAlloc and BadValue errors.

The *XGetModifierMapping()* function returns a pointer to a newly created **XModifierKeymap** structure that contains the keys being used as modifiers. The structure should be freed after use by calling **XFreeModifiermap**. If only zero values appear in the set for any modifier, that modifier is disabled.

The *XNewModifiermap()* function returns a pointer to **XModifierKeymap** structure for later use.

The *XInsertModifiermapEntry()* function adds the specified KeyCode to the set that controls the specified modifier and returns the resulting **XModifierKeymap** structure (expanded as needed).

The *XDeleteModifiermapEntry()* function deletes the specified KeyCode from the set that controls the specified modifier and returns a pointer to the resulting **XModifierKeymap** structure.

The *XFreeModifiermap()* function frees the specified **XModifierKeymap** structure.

STRUCTURES

The **XModifierKeymap** structure contains:

```
typedef struct {
    int max_keypermod;           This server's max number of keys per modifier
    KeyCode *modifiermap;       An 8 by max_keypermod array of the modifiers
} XModifierKeymap;
```

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives

can generate this error.

SEE ALSO

XFree(), *XSetPointerMapping()*

Xlib — C Language X Interface

NAME

XChangePointerControl(), XGetPointerControl() – control pointer

SYNOPSIS

```
XChangePointerControl (Display *display, Bool do_accel,
                      Bool do_threshold, int accel_numerator,
                      int accel_denominator, int threshold)
XGetPointerControl (Display *display, int *accel_numerator_return,
                  int *accel_denominator_return,
                  int *threshold_return)
```

ARGUMENTS

accel_denominator
Specifies the denominator for the acceleration multiplier.

accel_denominator_return
Returns the denominator for the acceleration multiplier.

accel_numerator
Specifies the numerator for the acceleration multiplier.

accel_numerator_return
Returns the numerator for the acceleration multiplier.

display Specifies the connection to the X server.

do_accel Specifies a Boolean value that controls whether the values for the *accel_numerator* or *accel_denominator* are used.

do_threshold
Specifies a Boolean value that controls whether the value for the threshold is used.

threshold
Specifies the acceleration threshold.

threshold_return
Returns the acceleration threshold.

DESCRIPTION

The *XChangePointerControl()* function defines how the pointing device moves. The acceleration, expressed as a fraction, is a multiplier for movement. For example, specifying 3/1 means the pointer moves three times as fast as normal. The fraction may be rounded arbitrarily by the X server. Acceleration only takes effect if the pointer moves more than threshold pixels at once and only applies to the amount beyond the value in the threshold argument. Setting a value to -1 restores the default. The values of the *do_accel* and *do_threshold* arguments must be True for the pointer values to be set, or the parameters are unchanged. Negative values (other than -1) generate a BadValue error, as does a zero value for the *accel_denominator* argument.

XChangePointerControl() can generate a BadValue error.

The *XGetPointerControl()* function returns the pointer's current acceleration multiplier and acceleration threshold.

DIAGNOSTICS

BadValue
Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives

can generate this error.

SEE ALSO

Xlib — C Language X Interface

NAME

XChangeSaveSet(), XAddToSaveSet(), XRemoveFromSaveSet() – change a client’s save set

SYNOPSIS

```
XChangeSaveSet (Display *display, Window w, int change_mode)
XAddToSaveSet (Display *display, Window w)
XRemoveFromSaveSet (Display *display, Window w)
```

ARGUMENTS

change_mode Specifies the mode. You can pass SetModeInsert or SetModeDelete.
display Specifies the connection to the X server.
w Specifies the window that you want to add or delete from the client’s save-set.

DESCRIPTION

Depending on the specified mode, *XChangeSaveSet()* either inserts or deletes the specified window from the client’s save-set. The specified window must have been created by some other client, or a BadMatch error results.

XChangeSaveSet() can generate BadMatch, BadValue, and BadWindow errors.

The *XAddToSaveSet()* function adds the specified window to the client’s save-set. The specified window must have been created by some other client, or a BadMatch error results.

XAddToSaveSet() can generate BadMatch and BadWindow errors.

The *XRemoveFromSaveSet()* function removes the specified window from the client’s save-set. The specified window must have been created by some other client, or a BadMatch error results.

XRemoveFromSaveSet() can generate BadMatch and BadWindow errors.

DIAGNOSTICS

BadMatch
 Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue
 Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument’s type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow
 A value for a Window argument does not name a defined Window.

SEE ALSO

XReparentWindow()
Xlib — C Language X Interface

NAME

XChangeWindowAttributes(), XSetWindowBackground(),
 XSetWindowBackgroundPixmap(), XSetWindowBorder(), XSetWindowBorderPixmap(),
 XSetWindowColormap() – change window attributes

SYNOPSIS

```
XChangeWindowAttributes (Display *display, Window w,
                        unsigned long valuemask,
                        XSetWindowAttributes *attributes)
XSetWindowBackground (Display *display, Window w,
                    unsigned long background_pixel)
XSetWindowBackgroundPixmap (Display *display, Window w,
                           Pixmap background_pixmap)
XSetWindowBorder (Display *display, Window w,
                 unsigned long border_pixel)
XSetWindowBorderPixmap (Display *display, Window w,
                       Pixmap border_pixmap)
XSetWindowColormap (Display *display, Window w,
                   Colormap colormap)
```

ARGUMENTS

attributes

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

background_pixel

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

background_pixmap

Specifies the background pixmap, ParentRelative, or None.

border_pixel

Specifies the entry in the colormap.

border_pixmap

Specifies the border pixmap or CopyFromParent.

display Specifies the connection to the X server.

valuemask

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

w

Specifies the window.

colormap

Specifies the colormap.

DESCRIPTION

Depending on the valuemask, the *XChangeWindowAttributes()* function uses the window attributes in the **XSetWindowAttributes** structure to change the specified window attributes. Changing the background does not cause the window contents to be changed. To repaint the window and its background, use *XClearWindow()*. Setting the border or changing the background such that the border tile origin changes causes the border to be repainted. Changing the background of a root window to None or ParentRelative restores the default background pixmap. Changing the border of a root window to CopyFromParent restores the default border pixmap. Changing the win-

gravity does not affect the current position of the window. Changing the backing-store of an obscured window to *WhenMapped* or *Always*, or changing the backing-planes, backing-pixel, or save-under of a mapped window may have no immediate effect. Changing the colormap of a window (that is, defining a new map, not changing the contents of the existing map) generates a *ColormapNotify* event. Changing the colormap of a visible window may have no immediate effect on the screen because the map may not be installed (see *XInstallColormap()*). Changing the cursor of a root window to *None* restores the default cursor. Whenever possible, you are encouraged to share colormaps.

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

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

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

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

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

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

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

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

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

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

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

XSetWindowColormap() can generate BadColor, BadMatch, and BadWindow errors.

DIAGNOSTICS

BadAccess

A client attempted to free a color map entry that it did not already allocate.

BadAccess

A client attempted to store into a read-only color map entry.

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadCursor

A value for a Cursor argument does not name a defined Cursor.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadMatch

An InputOnly window locks this attribute.

BadPixmap

A value for a Pixmap argument does not name a defined Pixmap.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XConfigureWindow(), *XCreateWindow()*, *XDestroyWindow()*, *XInstallColormap()*, *XMapWindow()*, *XRaiseWindow()*, *XUnmapWindow()*

Xlib — C Language X Interface

NAME

XCirculateEvent – CirculateNotify event structure

STRUCTURES

The structure for CirculateNotify events contains:

```
typedef struct {
    int type;                CirculateNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window event;
    Window window;
    int place;               PlaceOnTop, PlaceOnBottom
} XCCirculateEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *event* member is set either to the restacked window or to its parent, depending on whether StructureNotify or SubstructureNotify was selected. The *window* member is set to the window that was restacked. The *place* member is set to the window's position after the restack occurs and is either PlaceOnTop or PlaceOnBottom. If it is PlaceOnTop, the window is now on top of all siblings. If it is PlaceOnBottom, the window is now below all siblings.

SEE ALSO

XAnyEvent, XButtonEvent, XCreateWindowEvent, XCirculateRequestEvent, XColormapEvent, XConfigureEvent, XConfigureRequestEvent, XCrossingEvent, XDestroyWindowEvent, XErrorEvent, XExposeEvent, XFocusChangeEvent, XGraphicsExposeEvent, XGravityEvent, XKeymapEvent, XMapEvent, XMapRequestEvent, XPropertyEvent, XReparentEvent, XResizeRequestEvent, XSelectionClearEvent, XSelectionEvent, XSelectionRequestEvent, XUnmapEvent, XVisibilityEvent

Xlib — C Language X Interface

NAME

XCirculateRequestEvent – CirculateRequest event structure

STRUCTURES

The structure for CirculateRequest events contains:

```
typedef struct {
    int type;                CirculateRequest
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window parent;
    Window window;
    int place;               PlaceOnTop, PlaceOnBottom
} XCCirculateRequestEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *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.

SEE ALSO

XAnyEvent, *XButtonEvent*, *XCreateWindowEvent*, *XCirculateEvent*, *XColormapEvent*, *XConfigureEvent*, *XConfigureRequestEvent*, *XCrossingEvent*, *XDestroyWindowEvent*, *XErrorEvent*, *XExposeEvent*, *XFocusChangeEvent*, *XGraphicsExposeEvent*, *XGravityEvent*, *XKeymapEvent*, *XMapEvent*, *XMapRequestEvent*, *XPropertyEvent*, *XReparentEvent*, *XResizeRequestEvent*, *XSelectionClearEvent*, *XSelectionEvent*, *XSelectionRequestEvent*, *XUnmapEvent*, *XVisibilityEvent*

Xlib — C Language X Interface

NAME

XClearArea(), XClearWindow() – clear area or window

SYNOPSIS

```
XClearArea (Display *display, Window w, int x, y,
            unsigned int width, height, Bool exposures)
XClearWindow (Display *display, Window w)
```

ARGUMENTS

display Specifies the connection to the X server.

exposures Specifies a Boolean value that indicates if Expose events are to be generated.

w Specifies the window.

width

height Specify the width and height, which are the dimensions of the rectangle.

x

y Specify the x and y coordinates, which are relative to the origin of the window and specify the upper-left corner of the rectangle.

DESCRIPTION

The *XClearArea()* function paints a rectangular area in the specified window according to the specified dimensions with the window's background pixel or pixmap. The subwindow-mode effectively is ClipByChildren. If *width* is zero, it is replaced with the current width of the window minus *x*. If *height* is zero, it is replaced with the current height of the window minus *y*. If the window has a defined background tile, the rectangle clipped by any children is filled with this tile. If the window has background None, the contents of the window are not changed. In either case, if *exposures* is True, one or more Expose events are generated for regions of the rectangle that are either visible or are being retained in a backing store. If you specify a window whose class is InputOnly, a BadMatch error results.

XClearArea() can generate BadMatch, BadValue, and BadWindow errors.

The *XClearWindow()* function clears the entire area in the specified window and is equivalent to

```
XClearArea(display, w, 0, 0, 0, 0, False).
```

If the window has a defined background tile, the rectangle is tiled with a plane-mask of all ones and *GXcopy()* function. If the window has background None, the contents of the window are not changed. If you specify a window whose class is InputOnly, a BadMatch error results.

XClearWindow() can generate BadMatch and BadWindow errors.

DIAGNOSTICS

BadMatch

An InputOnly window is used as a Drawable.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XCopyArea()

Xlib — C Language X Interface

NAME

XClientMessageEvent – ColormapNotify event structure

STRUCTURES

The structure for ClientMessage events contains:

```
typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Window window;
    Atom message_type;
    int format;
    union {
        char b[20];
        short s[10];
        long l[5];
    } data;
} XClientMessageEvent;
```

ClientMessage
of last request processed by server
true if this came from a SendEvent request
Display the event was read from

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *message_type* member is set to an atom that indicates how the data should be interpreted by the receiving client. The *format* member is set to 8, 16, or 32 and specifies whether the data should be viewed as a list of bytes, shorts, or longs. The *data* member is a union that contains the members *b*, *s*, and *l*. The *b*, *s*, and *l* members represent data of 20 8-bit values, 10 16-bit values, and 5 32-bit values. Particular message types might not make use of all these values. The X server places no interpretation on the values in the *window*, *message_type*, or *data* members.

SEE ALSO

XAnyEvent, *XButtonEvent*, *XCreateWindowEvent*, *XCirculateEvent*,
XCirculateRequestEvent, *XColormapEvent*, *XConfigureEvent*, *XConfigureRequestEvent*,
XCrossingEvent, *XDestroyWindowEvent*, *XErrorEvent*, *XExposeEvent*,
XFocusChangeEvent, *XGraphicsExposeEvent*, *XGravityEvent*, *XKeyEvent*,
XMapEvent, *XMapRequestEvent*, *XPropertyEvent*, *XReparentEvent*,
XResizeRequestEvent, *XSelectionClearEvent*, *XSelectionEvent*, *XSelectionRequestEvent*,
XUnmapEvent, *XVisibilityEvent*

Xlib — C Language X Interface

NAME

XColormapEvent – ColormapNotify event structure

STRUCTURES

The structure for ColormapNotify events contains:

```
typedef struct {
    int type;                ColormapNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;
    Colormap colormap;       colormap or None
    Bool new;
    int state;               ColormapInstalled, ColormapUninstalled
} XColormapEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to **GraphicsExpose**. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *window* member is set to the window whose associated colormap is changed, installed, or uninstalled. For a colormap that is changed, installed, or uninstalled, the colormap member is set to the colormap associated with the window. For a colormap that is changed by a call to **XFreeColormap**, the colormap member is set to None. The *new* member is set to indicate whether the colormap for the specified window was changed or installed or uninstalled and can be True or False. If it is True, the colormap was changed. If it is False, the colormap was installed or uninstalled. The state member is always set to indicate whether the colormap is installed or uninstalled and can be **ColormapInstalled** or **ColormapUninstalled**.

SEE ALSO

XAnyEvent, XButtonEvent, XCreateWindowEvent, XCirculateEvent, XCirculateRequestEvent, XConfigureEvent, XConfigureRequestEvent, XCreateColormap, XCrossingEvent, XDestroyWindowEvent, XErrorEvent, XExposeEvent, XFocusChangeEvent, XGraphicsExposeEvent, XGravityEvent, XKeymapEvent, XMapEvent, XMapRequestEvent, XPropertyEvent, XReparentEvent, XResizeRequestEvent, XSelectionClearEvent, XSelectionEvent, XSelectionRequestEvent, XUnmapEvent, XVisibilityEvent

Xlib — C Language X Interface

NAME

XConfigureEvent – ConfigureNotify event structure

STRUCTURES

The structure for ConfigureNotify events contains:

```
typedef struct {
    int type;                ConfigureNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window event;
    Window window;
    int x, y;
    int width, height;
    int border_width;
    Window above;
    Bool override_redirect;
} XConfigureEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *event* member is set either to the reconfigured window or to its parent, depending on whether StructureNotify or SubstructureNotify was selected. The *window* member is set to the window whose size, position, border, and / or stacking order was changed.

The *x* and *y* members are set to the coordinates relative to the parent window's origin and indicate the position of the upper-left outside corner of the window. The width and height members are set to the inside size of the window, not including the border. The *border_width* member is set to the width of the window's border, in pixels.

The *above* member is set to the sibling window and is used for stacking operations. If the X server sets this member to None, the window whose state was changed is on the bottom of the stack with respect to sibling windows. However, if this member is set to a sibling window, the window whose state was changed is placed on top of this sibling window.

The *override_redirect* member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the *override_redirect* member is True.

SEE ALSO

XAnyEvent, *XButtonEvent*, *XCreateWindowEvent*, *XCirculateEvent*,
XCirculateRequestEvent, *XColormapEvent*, *XConfigureRequestEvent*, *XCrossingEvent*,
XDestroyWindowEvent, *XErrorEvent*, *XExposeEvent*, *XFocusChangeEvent*,

*XGraphicsExposeEvent, XGravityEvent, XKeymapEvent, XMapEvent,
XMapRequestEvent, XPropertyEvent, XReparentEvent, XResizeRequestEvent,
XSelectionClearEvent, XSelectionEvent, XSelectionRequestEvent, XUnmapEvent,
XVisibilityEvent*

Xlib — C Language X Interface

NAME

XConfigureRequestEvent – ConfigureRequest event structure

STRUCTURES

The structure for ConfigureRequest events contains:

```
typedef struct {
    int type;                ConfigureRequest
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window parent;
    Window window;
    int x, y;
    int width, height;
    int border_width;
    Window above;
    int detail;              Above, Below, TopIf, BottomIf, Opposite
    unsigned long value_mask;
} XConfigureRequestEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to **GraphicsExpose**. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *parent* member is set to the parent window. The *window* member is set to the window whose size, position, border width, and / or stacking order is to be reconfigured. The *value_mask* member indicates which components were specified in the ConfigureWindow protocol request. The corresponding values are reported as given in the request. The remaining values are filled in from the current geometry of the window, except in the case of above (sibling) and detail (stack-mode), which are reported as Above and None , respectively, if they are not given in the request.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*, *XCrossingEvent()*,
XDestroyWindowEvent(), *XErrorEvent()*, *XExposeEvent()*, *XFocusChangeEvent()*,
XGraphicsExposeEvent(), *XGravityEvent()*, *XKeymapEvent()*, *XMapEvent()*,
XMapRequestEvent(), *XPropertyEvent()*, *XReparentEvent()*, *XResizeRequestEvent()*,
XSelectionClearEvent(), *XSelectionEvent()*, *XSelectionRequestEvent()*, *XUnmapEvent()*,
XVisibilityEvent()

Xlib — C Language X Interface

NAME

XConfigureWindow(), XMoveWindow(), XResizeWindow(), XMoveResizeWindow(), XSetWindowBorderWidth(), XWindowChanges() – configure windows and window changes structure

SYNOPSIS

```
XConfigureWindow (Display *display, Window w,
                  unsigned int value_mask, XWindowChanges *values)
XMoveWindow (Display *display, Window w, int x, int y)
XResizeWindow (Display *display, Window w, unsigned int width,
               unsigned int height)
XMoveResizeWindow (Display *display, Window w, int x,
                  int y, unsigned int width, unsigned int height)
XSetWindowBorderWidth (Display *display, Window w,
                       unsigned int bwidth)
```

ARGUMENTS

display Specifies the connection to the X server.

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

values Specifies the **XWindowChanges** structure.

w Specifies the window to be reconfigured, moved, or resized..

bwidth Specifies the width of the window border.

width

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

x

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

DESCRIPTION

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

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

XConfigureWindow() can generate BadMatch, BadValue, and BadWindow errors.

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

Expose() events. Moving a mapped window generates *Expose()* events on any formerly obscured windows.

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

XMoveWindow() can generate a *BadWindow* error.

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

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

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

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

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

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

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

XSetWindowBorderWidth() can generate a *BadWindow* error.

STRUCTURES

The **XWindowChanges** structure contains:

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

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

Values

```
typedef struct {
    int x, y;
```

```

    int width, height;
    int border_width;
    Window sibling;
    int stack_mode;
} XWindowChanges;

```

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

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

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

DIAGNOSTICS

BadMatch

An *InputOnly* window is used as a *Drawable*.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a *Window* argument does not name a defined *Window*.

SEE ALSO

XChangeWindowAttributes(), *XCreateWindow()*, *XDestroyWindow()*, *XMapWindow()*, *XRaiseWindow()*, *XUnmapWindow()*

Xlib — C Language X Interface

NAME

XCopyArea(), XCopyPlane() – copy areas

SYNOPSIS

```
XCopyArea (Display *display, Drawable src, Drawable dest,
           GC gc, int src_x, int src_y, unsigned int width,
           unsigned int height, int dest_x, int dest_y)
XCopyPlane (Display *display, Drawable src, Drawable dest,
            GC gc, int src_x, int src_y, unsigned int width,
            unsigned int height, int dest_x, int dest_y,
            unsigned long plane)
```

ARGUMENTS

dest_x
dest_y Specify the x and y coordinates, which are relative to the origin of the destination rectangle and specify its upper-left corner.

display Specifies the connection to the X server.

gc Specifies the GC.

plane Specifies the bit plane. You must set exactly one bit to 1.

src
dest Specify the source and destination rectangles to be combined.

src_x
src_y Specify the x and y coordinates, which are relative to the origin of the source rectangle and specify its upper-left corner.

width
height Specify the width and height, which are the dimensions of both the source and destination rectangles.

DESCRIPTION

The *XCopyArea()* function combines the specified rectangle of *src* with the specified rectangle of *dest*. The drawables must have the same root and depth, or a *BadMatch* error results.

If regions of the source rectangle are obscured and have not been retained in backing store or if regions outside the boundaries of the source drawable are specified, those regions are not copied. Instead, the following occurs on all corresponding destination regions that are either visible or are retained in backing store. If the destination is a window with a background other than *None*, corresponding regions of the destination are tiled with that background (with plane-mask of all ones and *GXcopy()* function). Regardless of tiling or whether the destination is a window or a pixmap, if *graphics-exposures* is *True*, then *GraphicsExpose* events for all corresponding destination regions are generated. If *graphics-exposures* is *True* but no *GraphicsExpose* events are generated, a *NoExpose* event is generated. Note that by default *graphics-exposures* is *True* in new GCs.

This function uses these GC components: function, plane-mask, subwindow-mode, *graphics-exposures*, clip-x-origin, clip-y-origin, and clip-mask.

XCopyArea() can generate *BadDrawable*, *BadGC*, and *BadMatch* errors.

The *XCopyPlane()* function uses a single bit plane of the specified source rectangle combined with the specified GC to modify the specified rectangle of *dest*. The drawables must have the same root but need not have the same depth. If the drawables

do not have the same root, a BadMatch error results. If plane does not have exactly one bit set to 1 and the values of planes must be less than 2^n , where n is the depth of *src*, a BadValue error results.

Effectively, *XCopyPlane()* forms a pixmap of the same depth as the rectangle of *dest* and with a size specified by the source region. It uses the foreground / background pixels in the GC (foreground everywhere the bit plane in *src* contains a bit set to 1, background everywhere the bit plane in *src* contains a bit set to 0) and the equivalent of a CopyArea protocol request is performed with all the same exposure semantics. This can also be thought of as using the specified region of the source bit plane as a stipple with a fill-style of FillOpaqueStippled for filling a rectangular area of the destination.

This function uses these GC components: function, plane-mask, foreground, background, subwindow-mode, graphics-exposures, clip-x-origin, clip-y-origin, and clip-mask.

XCopyPlane() can generate BadDrawable, BadGC, BadMatch, and BadValue errors.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XClearArea()

Xlib — C Language X Interface

NAME

XCreateColormap(), XCopyColormapAndFree(), XFreeColormap(), XColor() – create, copy, or destroy colormaps and color structure

SYNOPSIS

```
Colormap XCreateColormap (Display *display, Window w,
                          Visual *visual, int alloc)
Colormap XCopyColormapAndFree (Display *display,
                              Colormap colormap)
XFreeColormap (Display *display, Colormap colormap)
```

ARGUMENTS

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

colormap Specifies the colormap that you want to create, copy, set, or destroy.

display Specifies the connection to the X server.

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

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

DESCRIPTION

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

The initial values of the colormap entries are undefined for the visual classes GrayScale, PseudoColor, and DirectColor. For StaticGray, StaticColor, and TrueColor, the entries have defined values, but those values are specific to the visual and are not defined by X. For StaticGray, StaticColor, and TrueColor, *alloc* must be AllocNone, or a BadMatch error results. For the other visual classes, if *alloc* is AllocNone, the colormap initially has no allocated entries, and clients can allocate them. For information about the visual types, see section 3.1.

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

XCreateColormap() can generate BadAlloc, BadMatch, BadValue, and BadWindow errors.

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

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

XCopYColormapAndFree() can generate BadAlloc and BadColor errors.

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

XFreeColormap() can generate a BadColor error.

STRUCTURES

The **XColor** structure contains:

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

```
} XColor;
```

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

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocColor(), *XChangeWindowAttributes()*, *XCreateWindow()*, *XQueryColor()*, *XStoreColors()*

NAME

XCreateFontCursor(), XCreatePixmapCursor(), XCreateGlyphCursor() – create cursors

SYNOPSIS

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

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

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

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

ARGUMENTS

background_color

Specifies the RGB values for the background of the source.

display Specifies the connection to the X server.

foreground_color

Specifies the RGB values for the foreground of the source.

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

mask_char

Specifies the glyph character for the mask.

mask_font

Specifies the font for the mask glyph or None.

shape Specifies the shape of the cursor.

source Specifies the shape of the source cursor.

source_char

Specifies the character glyph for the source.

source_font

Specifies the font for the source glyph.

x

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

DESCRIPTION

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

The hotspot comes from the information stored in the cursor font. The initial colors of a cursor are a black foreground and a white background (see *XRecolorCursor()*).

XCreateFontCursor() can generate BadAlloc and BadValue errors.

The *XCreatePixmapCursor()* function creates a cursor and returns the cursor ID associated with it. The foreground and background RGB values must be specified using *foreground_color* and *background_color*, even if the X server only has a StaticGray or GrayScale screen. The foreground color is used for the pixels set to 1 in the source, and

the background color is used for the pixels set to 0. Both source and mask, if specified, must have depth one (or a BadMatch error results) but can have any root. The *mask* argument defines the shape of the cursor. The pixels set to 1 in the *mask* define which source pixels are displayed, and the pixels set to 0 define which pixels are ignored. If no *mask* is given, all pixels of the source are displayed. The *mask*, if present, must be the same size as the pixmap defined by the source argument, or a BadMatch error results. The hotspot must be a point within the source, or a BadMatch error results.

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

XCreatePixmapCursor() can generate BadAlloc and BadPixmap errors.

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

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

XCreateGlyphCursor() can generate BadAlloc, BadFont, and BadValue errors.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadFont

A value for a Font or GContext argument does not name a defined Font.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadPixmap

A value for a Pixmap argument does not name a defined Pixmap.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XDefineCursor(), *XLoadFont()*, *XRecolorCursor()*

Xlib — C Language X Interface

NAME

XCreateFontSet(), XFreeFontSet() – create and free an international text drawing font set

SYNOPSIS

```
XFontSet XCreateFontSet (Display *display,
                        char *base_font_name_list,
                        char ***missing_charset_list_return,
                        int *missing_charset_count_return,
                        char **def_string_return)
void XFreeFontSet (Display *display, XFontSet font_set)
```

ARGUMENTS

display Specifies the connection to the X server.
base_font_name_list
 Specifies the base font names.
def_string_return
 Returns the string drawn for missing charsets.
font_set Specifies the font set.
missing_charset_count_return
 Returns the number of missing charsets.
missing_charset_list_return
 Returns the missing charsets.

DESCRIPTION

The *XCreateFontSet()* function creates a font set for the specified display. The font set is bound to the current locale when *XCreateFontSet()* is called. The *font_set* may be used in subsequent calls to obtain font and character information, and to image text in the locale of the *font_set*.

The *base_font_name_list* argument is a list of base font names which Xlib uses to load the fonts needed for the locale. The base font names are a comma-separated list. The string is null-terminated, and is assumed to be in the Host Portable Character Encoding; otherwise, the result is implementation dependent. Whitespace immediately on either side of a separating comma is ignored.

Use of XLFD font names permits Xlib to obtain the fonts needed for a variety of locales from a single locale-independent base font name. The single base font name should name a family of fonts whose members are encoded in the various charsets needed by the locales of interest.

An XLFD base font name can explicitly name a charset needed for the locale. This allows the user to specify an exact font for use with a charset required by a locale, fully controlling the font selection.

If a base font name is not an XLFD name, Xlib will attempt to obtain an XLFD name from the font properties for the font. If this action is successful in obtaining an XLFD name, the *XBaseFontNameListOfFontSet()* function will return this XLFD name instead of the client-supplied name.

The following algorithm is used to select the fonts that will be used to display text with the *XFontSet()*:

For each font charset required by the locale, the base font name list is searched for the first one of the following cases that names a set of fonts that exist at the server:

1. The first XLFD-conforming base font name that specifies the required charset or a superset of the required charset in its *CharSetRegistry()* and *CharSetEncoding()* fields. The implementation may use a base font name whose specified charset is a superset of the required charset, for example, an ISO8859-1 font for an ASCII charset.
2. The first set of one or more XLFD-conforming base font names that specify one or more charsets that can be remapped to support the required charset. The Xlib implementation may recognize various mappings from a required charset to one or more other charsets, and use the fonts for those charsets. For example, JIS Roman is ASCII with tilde and backslash replaced by yen and overbar; Xlib may load an ISO8859-1 font to support this character set, if a JIS Roman font is not available.
3. The first XLFD-conforming font name, or the first non-XLFD font name for which an XLFD font name can be obtained, combined with the required charset (replacing the *CharSetRegistry()* and *CharSetEncoding()* fields in the XLFD font name). As in case 1, the implementation may use a charset which is a superset of the required charset.
4. The first font name that can be mapped in some implementation-dependent manner to one or more fonts that support imaging text in the charset.

For example, assume a locale required the charsets:

```
ISO8859-1
JISX0208.1983
JISX0201.1976
GB2312-1980.0
```

The user could supply a *base_font_name_list* which explicitly specifies the charsets, insuring that specific fonts get used if they exist:

```
"-JIS-Fixed-Medium-R-Normal--26-180-100-100-C-240-JISX0208.1983-0,\
-JIS-Fixed-Medium-R-Normal--26-180-100-100-C-120-JISX0201.1976-0,\
-GB-Fixed-Medium-R-Normal--26-180-100-100-C-240-GB2312-1980.0,\
-Adobe-Courier-Bold-R-Normal--25-180-75-75-M-150-ISO8859-1"
```

Or he could supply a *base_font_name_list* which omits the charsets, letting Xlib select font charsets required for the locale:

```
"-JIS-Fixed-Medium-R-Normal--26-180-100-100-C-240,\
-JIS-Fixed-Medium-R-Normal--26-180-100-100-C-120,\
-GB-Fixed-Medium-R-Normal--26-180-100-100-C-240,\
-Adobe-Courier-Bold-R-Normal--25-180-100-100-M-150"
```

Or he could simply supply a single base font name which allows Xlib to select from all available fonts which meet certain minimum XLFD property requirements:

```
"-*-*-*-R-Normal--*-180-100-100-*-*"
```

If *XCreateFontSet()* is unable to create the font set, either because there is insufficient memory or because the current locale is not supported, *XCreateFontSet()* returns NULL, *missing_charset_list_return* is set to NULL, and *missing_charset_count_return* is set to zero. If fonts exist for all of the charsets required by the current locale, *XCreateFontSet()* returns a valid *XFontSet()*, *missing_charset_list_return* is set to NULL, and *missing_charset_count_return* is set to zero.

If no font exists for one or more of the required charsets, *XCreateFontSet()* sets *missing_charset_list_return* to a list of one or more null-terminated charset names for which no font exists, and sets *missing_charset_count_return* to the number of missing fonts. The charsets are from the list of the required charsets for the encoding of the locale, and do not include any charsets to which Xlib may be able to remap a required charset.

If no font exists for any of the required charsets, or if the locale definition in Xlib requires that a font exist for a particular charset and a font is not found for that charset, *XCreateFontSet()* returns NULL. Otherwise, *XCreateFontSet()* returns a valid *XFontSet()* to *font_set*.

When an Xmb/wc drawing or measuring function is called with an *XFontSet()* that has missing charsets, some characters in the locale will not be drawable. If *def_string_return* is non-NULL, *XCreateFontSet()* returns a pointer to a string which represents the glyph(s) which are drawn with this *XFontSet()* when the charsets of the available fonts do not include all font glyph(s) required to draw a codepoint. The string does not necessarily consist of valid characters in the current locale and is not necessarily drawn with the fonts loaded for the font set, but the client can draw and measure the “default glyphs” by including this string in a string being drawn or measured with the *XFontSet()*.

If the string returned to *def_string_return* is the empty string (“”), no glyphs are drawn, and the escapement is zero. The returned string is null-terminated. It is owned by Xlib and should not be modified or freed by the client. It will be freed by a call to *XFreeFontSet()* with the associated *XFontSet()*. Until freed, its contents will not be modified by Xlib.

The client is responsible for constructing an error message from the missing charset and default string information, and may choose to continue operation in the case that some fonts did not exist.

The returned *XFontSet()* and missing charset list should be freed with *XFreeFontSet()* and *XFreeStringList()*, respectively. The client-supplied *base_font_name_list* may be freed by the client after calling *XCreateFontSet()*.

The *XFreeFontSet()* function frees the specified font set. The associated base font name list, font name list, *XFontStruct()* list, and *XFontSetExtents()*, if any, are freed.

SEE ALSO

XExtentsOfFontSet(), *XFontsOfFontSet()*, *XFontSetExtents()*

Xlib — C Language X Interface

NAME

XCreateGC(), XCopyGC(), XChangeGC(), XGetGCValues(), XFreeGC(), XGContextFromGC(), XGCValues() – create or free graphics contexts and graphics context structure

SYNOPSIS

```
GC XCreateGC (Display *display, Drawable d,
              unsigned long valuemask, XGCValues * values)
XCopyGC (Display *display, GC src, GC dest,
         unsigned long valuemask)
XChangeGC (Display *display, GC gc, unsigned long valuemask,
          XGCValues * values)
Status XGetGCValues (Display *display, GC gc,
                   unsigned long valuemask,
                   XGCValues *values_return)
XFreeGC (Display *display, GC gc)
GContext XGContextFromGC (GC gc)
```

ARGUMENTS

d Specifies the drawable.
dest Specifies the destination GC.
display Specifies the connection to the X server.
gc Specifies the GC.
src Specifies the components of the source GC.
valuemask Specifies which components in the GC are to be set, copied, changed, or returned. This argument is the bitwise inclusive OR of zero or more of the valid GC component mask bits.
values Specifies any values as specified by the valuemask.
values_return Returns the GC values in the specified **XGCValues** structure.

DESCRIPTION

The *XCreateGC()* function creates a graphics context and returns a GC. The GC can be used with any destination drawable having the same root and depth as the specified drawable. Use with other drawables results in a BadMatch error.

XCreateGC() can generate BadAlloc, BadDrawable, BadFont, BadMatch, BadPixmap, and BadValue errors.

The *XCopyGC()* function copies the specified components from the source GC to the destination GC. The source and destination GCs must have the same root and depth, or a BadMatch error results. The valuemask specifies which component to copy, as for *XCreateGC()*.

XCopyGC() can generate BadAlloc, BadGC, and BadMatch errors.

The *XChangeGC()* function changes the components specified by valuemask for the specified GC. The values argument contains the values to be set. The values and restrictions are the same as for *XCreateGC()*. Changing the clip-mask overrides any previous XSetClipRectangles request on the context. Changing the dash-offset or dash-list overrides any previous XSetDashes request on the context. The order in which components are verified and altered is server-dependent. If an error is generated, a

subset of the components may have been altered.

XChangeGC() can generate *BadAlloc*, *BadFont*, *BadGC*, *BadMatch*, *BadPixmap*, and *BadValue* errors.

The *XGetGCValues()* function returns the components specified by *valuemask* for the specified GC. If the *valuemask* contains a valid set of GC mask bits (*GCFunction*, *GCPlaneMask*, *GCForeground*, *GCBgground*, *GCLineWidth*, *GCLineStyle*, *GCCapStyle*, *GCJoinStyle*, *GCFillStyle*, *GCFillRule*, *GCTile*, *GCStipple*, *GCTileStipXOrigin*, *GCTileStipYOrigin*, *GCFont*, *GCSubwindowMode*, *GCGraphicsExposures*, *GCclipXOrigin*, *GCclipYOrigin*, *GCDashOffset*, or *GCArcMode*) and no error occur, *XGetGCValues()* sets the requested components in *values_return* and returns a nonzero status. Otherwise, it returns a zero status. Note that the clip-mask and dash-list (represented by the *GCclipMask* and *GCDashList* bits, respectively, in the *valuemask*) cannot be requested. Also note that an invalid resource ID (with one or more of the three most-significant bits set to one) will be returned for *GCFont*, *GCTile*, and *GCStipple* if the component has never been explicitly set by the client.

The *XFreeGC()* function destroys the specified GC as well as all the associated storage.

XFreeGC() can generate a *BadGC* error.

STRUCTURES

The **XGCValues** structure contains:

GC attribute value mask bits

```
#define GCFunction          (1L<<0)
#define GCPlaneMask        (1L<<1)
#define GCForeground        (1L<<2)
#define GCBgground         (1L<<3)
#define GCLineWidth         (1L<<4)
#define GCLineStyle         (1L<<5)
#define GCCapStyle          (1L<<6)
#define GCJoinStyle         (1L<<7)
#define GCFillStyle         (1L<<8)
#define GCFillRule          (1L<<9)
#define GCTile              (1L<<10)
#define GCStipple           (1L<<11)
#define GCTileStipXOrigin   (1L<<12)
#define GCTileStipYOrigin   (1L<<13)
#define GCFont              (1L<<14)
#define GCSubwindowMode     (1L<<15)
#define GCGraphicsExposures (1L<<16)
#define GCclipXOrigin       (1L<<17)
#define GCclipYOrigin       (1L<<18)
#define GCclipMask          (1L<<19)
#define GCDashOffset        (1L<<20)
#define GCDashList          (1L<<21)
```

```
#define GCArcMode (1L<<22)
```

Values

```
typedef struct {
    int function;                logical operation
    unsigned long plane_mask;    plane mask
    unsigned long foreground;    foreground pixel
    unsigned long background;    background pixel
    int line_width;              line width (in pixels)
    int line_style;              LineSolid, LineOnOffDash, LineDoubleDash
    int cap_style;               CapNotLast, CapButt, CapRound, CapProjecting
    int join_style;              JoinMiter, JoinRound, JoinBevel
    int fill_style;              FillSolid, FillTiled, FillStippled FillOpaqueStippled
    int fill_rule;               EvenOddRule, WindingRule
    int arc_mode;                ArcChord, ArcPieSlice
    Pixmap tile;                 tile pixmap for tiling operations
    Pixmap stipple;              stipple 1 plane pixmap for stippling
    int ts_x_origin;             offset for tile or stipple operations
    int ts_y_origin;
    Font font;                   default text font for text operations
    int subwindow_mode;          ClipByChildren, IncludeInferiors
    Bool graphics_exposures;     boolean, should exposures be generated
    int clip_x_origin;           origin for clipping
    int clip_y_origin;
    Pixmap clip_mask;            bitmap clipping; other calls for rects
    int dash_offset;             patterned / dashed line information
    char dashes;
} XGCValues;
```

The function attributes of a GC are used when you update a section of a drawable (the destination) with bits from somewhere else (the source). The function in a GC defines how the new destination bits are to be computed from the source bits and the old destination bits. **GXcopy** is typically the most useful because it will work on a color display, but special applications may use other functions, particularly in concert with particular planes of a color display. The 16 GC functions, defined in <X11/X.h>, are:

Function Name	Value	Operation
GXclear	0x0	0
GXand	0x1	src AND dst
GXandReverse	0x2	src AND NOT dst
GXcopy	0x3	src
GXandInverted	0x4	(NOT src) AND dst
GXnoop	0x5	dst
GXxor	0x6	src XOR dst
GXor	0x7	src OR dst
GXnor	0x8	(NOT src) AND (NOT dst)
GXequiv	0x9	(NOT src) XOR dst
GXinvert	0xa	NOT dst

Function Name	Value	Operation
GXorReverse	0xb	src OR (NOT dst)
GXcopyInverted	0xc	NOT src
GXorInverted	0xd	(NOT src) OR dst
GXnand	0xe	(NOT src) OR (NOT dst)
GXset	0xf	1

Many graphics operations depend on either pixel values or planes in a GC. The planes attribute is of type long, and it specifies which planes of the destination are to be modified, one bit per plane. A monochrome display has only one plane and will be the least-significant bit of the word. As planes are added to the display hardware, they will occupy more significant bits in the plane mask.

In graphics operations, given a source and destination pixel, the result is computed bitwise on corresponding bits of the pixels. That is, a Boolean operation is performed in each bit plane. The *plane_mask* restricts the operation to a subset of planes. A macro constant AllPlanes can be used to refer to all planes of the screen simultaneously. The result is computed by the following:

((src FUNC dst) AND plane-mask) OR (dst AND (NOT plane-mask))

Range checking is not performed on the values for *foreground*, *background*, or *plane_mask*. They are simply truncated to the appropriate number of bits. The line-width is measured in pixels and either can be greater than or equal to one (wide line) or can be the special value zero (thin line).

Wide lines are drawn centered on the path described by the graphics request. Unless otherwise specified by the join-style or cap-style, the bounding box of a wide line with endpoints [x1, y1], [x2, y2] and width w is a rectangle with vertices at the following real coordinates:

$[x1 - (w * \sin/2), y1 + (w * \cos/2)], [x1 + (w * \sin/2), y1 - (w * \cos/2)],$
 $[x2 - (w * \sin/2), y2 + (w * \cos/2)], [x2 + (w * \sin/2), y2 - (w * \cos/2)]$

Here *sn* is the sine of the angle of the line, and *cs* is the cosine of the angle of the line. A pixel is part of the line and so is drawn if the center of the pixel is fully inside the bounding box (which is viewed as having infinitely thin edges). If the center of the pixel is exactly on the bounding box, it is part of the line if and only if the interior is immediately to its right (x increasing direction). Pixels with centers on a horizontal edge are a special case and are part of the line if and only if the interior or the boundary is immediately below (y increasing direction) and the interior or the boundary is immediately to the right (x increasing direction).

Thin lines (zero line-width) are one-pixel-wide lines drawn using an unspecified, device-dependent algorithm. There are only two constraints on this algorithm.

1. If a line is drawn unclipped from [x1,y1] to [x2,y2] and if another line is drawn unclipped from [x1+dx,y1+dy] to [x2+dx,y2+dy], a point [x,y] is touched by drawing the first line if and only if the point [x+dx,y+dy] is touched by drawing the second line.
2. The effective set of points comprising a line cannot be affected by clipping. That is, a point is touched in a clipped line if and only if the point lies inside the clipping region and the point would be touched by the line when drawn unclipped.

A wide line drawn from [x1,y1] to [x2,y2] always draws the same pixels as a wide line drawn from [x2,y2] to [x1,y1], not counting cap-style and join-style. It is recommended that this property be true for thin lines, but this is not required. A line-width of zero may differ from a line-width of one in which pixels are drawn. This permits the use of many manufacturers' line drawing hardware, which may run many times faster than the more precisely specified wide lines.

In general, drawing a thin line will be faster than drawing a wide line of width one. However, because of their different drawing algorithms, thin lines may not mix well aesthetically with wide lines. If it is desirable to obtain precise and uniform results across all displays, a client should always use a line-width of one rather than a line-width of zero.

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

LineSolid	The full path of the line is drawn.
LineDoubleDash	The full path of the line is drawn, but the even dashes are filled differently than the odd dashes (see fill-style) with CapButt style used where even and odd dashes meet.
LineOnOffDash	Only the even dashes are drawn, and cap-style applies to all internal ends of the individual dashes, except CapNotLast is treated as CapButt.

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

CapNotLast	This is equivalent to CapButt except that for a line-width of zero the final endpoint is not drawn.
CapButt	The line is square at the endpoint (perpendicular to the slope of the line) with no projection beyond.
CapRound	The line has a circular arc with the diameter equal to the line-width, centered on the endpoint. (This is equivalent to CapButt for line-width of zero).
CapProjecting	The line is square at the end, but the path continues beyond the endpoint for a distance equal to half the line-width. (This is equivalent to CapButt for line-width of zero).

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

JoinMiter	The outer edges of two lines extend to meet at an angle. However, if the angle is less than 11 degrees, then a JoinBevel join-style is used instead.
JoinRound	The corner is a circular arc with the diameter equal to the line-width, centered on the joinpoint.
JoinBevel	The corner has CapButt endpoint styles with the triangular notch filled.

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

CapNotLast	thin	The results are device-dependent, but the desired effect is that nothing is drawn. The results are device-dependent, but the desired effect is that a
------------	------	--

		single pixel is drawn.
CapButt	thin	
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 ($x_1=x_2$, $y_1=y_2$), when the join-style is applied at one or both endpoints, the effect is as if the line was removed from the overall path. However, if the total path consists of or is reduced to a single point joined with itself, the effect is the same as when the cap-style is applied at both endpoints.

The tile / stipple represents an infinite 2D plane, with the tile / stipple replicated in all dimensions. When that plane is superimposed on the drawable for use in a graphics operation, the upper left corner of some instance of the tile / stipple is at the coordinates within the drawable specified by the tile / stipple origin. The tile / stipple and clip origins are interpreted relative to the origin of whatever destination drawable is specified in a graphics request. The tile pixmap must have the same root and depth as the GC, or a BadMatch error results. The stipple pixmap must have depth one and must have the same root as the GC, or a BadMatch error results. For stipple operations where the fill-style is FillStippled but not FillOpaqueStippled, the stipple pattern is tiled in a single plane and acts as an additional clip mask to be ANDed with the clip-mask. Although some sizes may be faster to use than others, any size pixmap can be used for tiling or stippling.

The fill-style defines the contents of the source for line, text, and fill requests. For all text and fill requests (for example, XDrawText, XDrawText16, XFillRectangle, XFillPolygon, and XFillArc); for line requests with line-style LineSolid (for example, XDrawLine, XDrawSegments, XDrawRectangle, XDrawArc); and for the even dashes for line requests with line-style LineOnOffDash or LineDoubleDash, the following apply:

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

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

FillSolid	Background
FillTiled	Same as for even dashes
FillOpaqueStippled	Same as for even dashes

FillStippled Background masked by stipple

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

For optimum performance, you should draw as much as possible with the same GC (without changing its components). The costs of changing GC components relative to using different GCs depend upon the display hardware and the server implementation. It is quite likely that some amount of GC information will be cached in display hardware and that such hardware can only cache a small number of GCs.

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

The clip-mask restricts writes to the destination drawable. If the clip-mask is set to a pixmap, it must have depth one and have the same root as the GC, or a *BadMatch* error results. If clip-mask is set to *None*, the pixels are always drawn regardless of the clip origin. The clip-mask also can be set by calling the *XSetClipRectangles()* or *XSetRegion()* functions. Only pixels where the clip-mask has a bit set to 1 are drawn. Pixels are not drawn outside the area covered by the clip-mask or where the clip-mask has a bit set to 0. The clip-mask affects all graphics requests. The clip-mask does not clip sources. The clip-mask origin is interpreted relative to the origin of whatever destination drawable is specified in a graphics request.

You can set the subwindow-mode to *ClipByChildren()* or *IncludeInferiors()*. For *ClipByChildren()*, both source and destination windows are additionally clipped by all viewable *InputOutput()* children. For *IncludeInferiors()*, neither source nor destination window is clipped by inferiors. This will result in including subwindow contents in the source and drawing through subwindow boundaries of the destination. The use of *IncludeInferiors()* on a window of one depth with mapped inferiors of differing depth is not illegal, but the semantics are undefined by the core protocol.

The fill-rule defines what pixels are inside (drawn) for paths given in *XFillPolygon()* requests and can be set to *EvenOddRule()* or *WindingRule()*. For *EvenOddRule()*, a point is inside if an infinite ray with the point as origin crosses the path an odd number of times. For *WindingRule()*, a point is inside if an infinite ray with the point as origin crosses an unequal number of clockwise and counterclockwise directed path segments. A clockwise directed path segment is one that crosses the ray from left to right as observed from the point. A counterclockwise segment is one that crosses the ray from right to left as observed from the point. The case where a directed line segment is coincident with the ray is uninteresting because you can simply choose a different ray that is not coincident with a segment.

For both *EvenOddRule()* and *WindingRule()*, a point is infinitely small, and the path is an infinitely thin line. A pixel is inside if the center point of the pixel is inside and the center point is not on the boundary. If the center point is on the boundary, the pixel is inside if and only if the polygon interior is immediately to its right (x increasing direction). Pixels with centers on a horizontal edge are a special case and are inside if and only if the polygon interior is immediately below (y increasing direction).

The arc-mode controls filling in the *XFillArcs()* function and can be set to *ArcPieSlice* or *ArcChord*. For *ArcPieSlice*, the arcs are pie-slice filled. For *ArcChord*, the arcs are chord filled.

The graphics-exposure flag controls *GraphicsExpose* event generation for *XCopyArea()* and *XCopyPlane()* requests (and any similar requests defined by extensions).

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadFont

A value for a Font or GContext argument does not name a defined Font.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An *InputOnly()* window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadPixmap

A value for a Pixmap argument does not name a defined Pixmap.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

AllPlanes(), *XCopyArea()*, *XCreateRegion()*, *XDrawArc()*, *XDrawLine()*, *XDrawRectangle()*, *XDrawText()*, *XFillRectangle()*, *XQueryBestSize()*, *XSetArcMode()*, *XSetClipOrigin()*, *XSetFillStyle()*, *XSetFont()*, *XSetLineAttributes()*, *XSetState()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XCreateIC(), XDestroyIC(), XIMOfIC() – create, destroy, and obtain the input method of an input context

SYNOPSIS

```
XIC XCreateIC (XIM im, ...)  
void XDestroyIC (XIC ic)  
XIM XIMOfIC (XIC ic)
```

ARGUMENTS

ic Specifies the input context.
im Specifies the input method.
 ... Specifies the variable length argument list to set XIC values.

DESCRIPTION

The *XCreateIC()* function creates a context within the specified input method.

Some of the arguments are mandatory at creation time, and the input context will not be created if they are not provided. Those arguments are the input style and the set of text callbacks (if the input style selected requires callbacks). All other input context values can be set later.

XCreateIC() returns a NULL value if no input context could be created. A NULL value could be returned for any of the following reasons:

- A required argument was not set.
- A read-only argument was set (for example, XNFilterEvents).
- The argument name is not recognized.
- The input method encountered an input method implementation dependent error.

The *XCreateIC()* can generate BadAtom, BadColor, BadPixmap, and BadWindow errors.

XDestroyIC() destroys the specified input context.

The *XIMOfIC()* function returns the input method associated with the specified input context.

DIAGNOSTICS

BadAtom

A value for an Atom argument does not name a defined Atom.

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadPixmap

A value for a Pixmap argument does not name a defined Pixmap.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XOpenIM(), *XSetICFocus()*, *XSetICValues()*, *XmbResetIC()*

Xlib — C Language X Interface

NAME

XCreateImage(), XGetPixel(), XPutPixel(), XSubImage(), XAddPixel(), XDestroyImage() – image utilities

SYNOPSIS

```
XImage *XCreateImage (Display *display, Visual *visual,
                     unsigned int depth, int format,
                     int offset, char *data, unsigned int width,
                     unsigned int height, int bitmap_pad,
                     int bytes_per_line)
unsigned long XGetPixel (XImage *ximage, int x, int y)
XPutPixel (XImage *ximage, int x, int y, unsigned long pixel)
XImage *XSubImage (XImage *ximage, int x, int y,
                  unsigned int subimage_width,
                  unsigned int subimage_height)
XAddPixel (XImage *ximage, long value)
XDestroyImage (XImage * ximage)
```

ARGUMENTS

bitmap_pad Specifies the quantum of a scanline (8, 16, or 32). In other words, the start of one scanline is separated in client memory from the start of the next scanline by an integer multiple of this many bits.

bytes_per_line Specifies the number of bytes in the client image between the start of one scanline and the start of the next.

data Specifies the image data.

depth Specifies the depth of the image.

display Specifies the connection to the X server.

format Specifies the format for the image. You can pass XYBitmap, XYPixmap, or ZPixmap.

height Specifies the height of the image, in pixels.

offset Specifies the number of pixels to ignore at the beginning of the scanline.

pixel Specifies the new pixel value.

subimage_height Specifies the height of the new subimage, in pixels.

subimage_width Specifies the width of the new subimage, in pixels.

value Specifies the constant value that is to be added.

visual Specifies the **Visual** structure.

width Specifies the width of the image, in pixels.

ximage Specifies the image.

x

y Specify the x and y coordinates.

DESCRIPTION

The *XCreateImage()* function allocates the memory needed for an **XImage** structure for the specified display but does not allocate space for the image itself. Rather, it initializes the structure byte-order, bit-order, and bitmap-unit values from the display and returns a pointer to the **XImage** structure. The red, green, and blue mask values are defined for Z format images only and are derived from the **Visual** structure passed in. Other values

also are passed in. The *offset* permits the rapid displaying of the image without requiring each scanline to be shifted into position. If you pass a zero value in *bytes_per_line*, Xlib assumes that the scanlines are contiguous in memory and calculates the value of *bytes_per_line* itself.

Note that when the image is created using *XCreateImage()*, *XGetImage()*, or *XSubImage()*, the destroy procedure that the *XDestroyImage()* function calls frees both the image structure and the data pointed to by the image structure.

The basic functions used to get a pixel, set a pixel, create a subimage, and add a constant value to an image are defined in the image object. The functions in this section are really macro invocations of the functions in the image object and are defined in **<X11/Xutil.h>**.

The *XGetPixel()* function returns the specified pixel from the named image. The pixel value is returned in normalized format (that is, the least-significant byte of the long is the least-significant byte of the pixel). The image must contain the x and y coordinates.

The *XPutPixel()* function overwrites the pixel in the named image with the specified pixel value. The input pixel value must be in normalized format (that is, the least-significant byte of the long is the least-significant byte of the pixel). The image must contain the x and y coordinates.

The *XSubImage()* function creates a new image that is a subsection of an existing one. It allocates the memory necessary for the new **XImage** structure and returns a pointer to the new image. The data is copied from the source image, and the image must contain the rectangle defined by *x*, *y*, *subimage_width*, and *subimage_height*.

The *XAddPixel()* function adds a constant value to every pixel in an image. It is useful when you have a base pixel value from allocating color resources and need to manipulate the image to that form.

The *XDestroyImage()* function deallocates the memory associated with the **XImage** structure.

SEE ALSO

XPutImage()

Xlib — C Language X Interface

NAME

XCreatePixmap(), XFreePixmap() – create or destroy pixmaps

SYNOPSIS

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

ARGUMENTS

d Specifies which screen the pixmap is created on.
depth Specifies the depth of the pixmap.
display Specifies the connection to the X server.
pixmap Specifies the pixmap.
width
height Specify the width and height, which define the dimensions of the *pixmap*.

DESCRIPTION

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

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

XCreatePixmap() can generate BadAlloc, BadDrawable, and BadValue errors.

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

XFreePixmap() can generate a BadPixmap error.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.
BadDrawable A value for a Drawable argument does not name a defined Window or Pixmap.
BadPixmap A value for a Pixmap argument does not name a defined Pixmap.
BadValue Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XCopyArea()

Xlib — C Language X Interface

NAME

XCreateRegion(), XSetRegion(), XDestroyRegion() – create or destroy regions

SYNOPSIS

```
Region XCreateRegion ()  
XSetRegion (Display *display, GC gc, Region r)  
XDestroyRegion (Region r)
```

ARGUMENTS

display Specifies the connection to the X server.
gc Specifies the GC.
r Specifies the region.

DESCRIPTION

The *XCreateRegion()* function creates a new empty region.

The *XSetRegion()* function sets the clip-mask in the GC to the specified region. Once it is set in the GC, the region can be destroyed.

The *XDestroyRegion()* function deallocates the storage associated with a specified region.

SEE ALSO

XEmptyRegion(), *XIntersectRegion()*

Xlib — C Language X Interface

NAME

XCreateWindow(), XCreateSimpleWindow(), XSetWindowAttributes() – create windows and window attributes structure

SYNOPSIS

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

ARGUMENTS

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

background Specifies the background pixel value of the window.

border Specifies the border pixel value of the window.

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

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

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

display Specifies the connection to the X server.

parent Specifies the parent window.

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

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

width

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

x

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

DESCRIPTION

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

order with respect to siblings.

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

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

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

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

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

XCreateSimpleWindow() can generate BadAlloc, BadMatch, BadValue, and BadWindow errors.

STRUCTURES

The **XSetWindowAttributes** structure contains:

Window attribute value mask bits

```
#define CWBackPixmap      (1L<<0)
#define CWBackPixel      (1L<<1)
#define CWBorderPixmap   (1L<<2)
#define CWBorderPixel    (1L<<3)
#define CWBitGravity      (1L<<4)
#define CWWinGravity      (1L<<5)
#define CWBackingStore    (1L<<6)
#define CWBackingPlanes  (1L<<7)
#define CWBackingPixel    (1L<<8)
```

```
#define CWOVERRIDE_REDIRECT (1L<<9)
#define CWSAVE_UNDER        (1L<<10)
#define CWEVENT_MASK        (1L<<11)
#define CWDONT_PROPAGATE     (1L<<12)
#define CWCOLORMAP           (1L<<13)
#define CWCURSOR             (1L<<14)
```

Values

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

For a detailed explanation of the members of this structure, see *Xlib — C Language X Interface*.

DIAGNOSTICS**BadAlloc**

The server failed to allocate the requested resource or server memory.

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadCursor

A value for a Cursor argument does not name a defined Cursor.

BadMatch

The values do not exist for an InputOnly window.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadPixmap

A value for a Pixmap argument does not name a defined Pixmap.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

*XChangeWindowAttributes(), XConfigureWindow(), XDefineCursor(),
XDestroyWindow(), XMapWindow(), XRaiseWindow(), XUnmapWindow()*

Xlib — C Language X Interface

NAME

XCreateWindowEvent – CreateNotify event structure

STRUCTURES

The structure for CreateNotify events contains:

```
typedef struct {
    int type;                CreateNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window parent;           parent of the window
    Window window;           window id of window created
    int x, y;                window location
    int width, height;       size of window
    int border_width;        border width
    Bool override_redirect;  creation should be overridden
} XCreateWindowEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *parent* member is set to the created window's parent. The *window* member specifies the created window. The *x* and *y* members are set to the created window's coordinates relative to the parent window's origin and indicate the position of the upper-left outside corner of the created window. The *width* and *height* members are set to the inside size of the created window (not including the border) and are always nonzero. The *border_width* member is set to the width of the created window's border, in pixels. The *override_redirect* member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the *override_redirect* member is True.

SEE ALSO

XAnyEvent, *XButtonEvent*, *XCirculateEvent*, *XCirculateRequestEvent*,
XColormapEvent, *XConfigureEvent*, *XConfigureRequestEvent*, *XCrossingEvent*,
XDestroyWindowEvent, *XErrorEvent*, *XExposeEvent*, *XFocusChangeEvent*,
XGraphicsExposeEvent, *XGravityEvent*, *XKeymapEvent*, *XMapEvent*,
XMapRequestEvent, *XPropertyEvent*, *XReparentEvent*, *XResizeRequestEvent*,
XSelectionClearEvent, *XSelectionEvent*, *XSelectionRequestEvent*, *XUnmapEvent*,
XVisibilityEvent

Xlib — C Language X Interface

NAME

XCrossingEvent – EnterNotify and LeaveNotify event structure

STRUCTURES

The structure for EnterNotify and LeaveNotify events contains:

```
typedef struct {
    int type;                EnterNotify or LeaveNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;        true if this came from a SendEvent request
    Display *display;       Display the event was read from
    Window window;         "event" window reported relative to
    Window root;           root window that the event occurred on
    Window subwindow;      child window
    Time time;             milliseconds
    int x, y;              pointer x, y coordinates in event window
    int x_root, y_root;    coordinates relative to root
    int mode;              NotifyNormal, NotifyGrab, NotifyUngrab
    int detail;            NotifyAncestor, NotifyVirtual, NotifyInferior,
                        NotifyNonlinear, NotifyNonlinearVirtual
    Bool same_screen;      same screen flag
    Bool focus;            boolean focus
    unsigned int state;    key or button mask
} XCrossingEvent;
typedef XCrossingEvent XEnterWindowEvent;
typedef XCrossingEvent XLeaveWindowEvent;
```

When you receive these events, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *window* member is set to the window on which the EnterNotify or LeaveNotify event was generated and is referred to as the event window. This is the window used by the X server to report the event, and is relative to the root window on which the event occurred. The *root* member is set to the root window of the screen on which the event occurred.

For a LeaveNotify event, if a child of the event window contains the initial position of the pointer, the subwindow component is set to that child. Otherwise, the X server sets the subwindow member to None. For an EnterNotify event, if a child of the event window contains the final pointer position, the subwindow component is set to that child or None.

The *time* member is set to the time when the event was generated and is expressed in milliseconds. The *x* and *y* members are set to the coordinates of the pointer position in the event window. This position is always the pointer's final position, not its initial

position. If the event window is on the same screen as the root window, *x* and *y* are the pointer coordinates relative to the event window's origin. Otherwise, *x* and *y* are set to zero. The *x_root* and *y_root* members are set to the pointer's coordinates relative to the root window's origin at the time of the event.

The *same_screen* member is set to indicate whether the event window is on the same screen as the root window and can be either True or False. If True, the event and root windows are on the same screen. If False, the event and root windows are not on the same screen.

The *focus* member is set to indicate whether the event window is the focus window or an inferior of the focus window. The X server can set this member to either True or False. If True, the event window is the focus window or an inferior of the focus window. If False, the event window is not the focus window or an inferior of the focus window.

The *state* member is set to indicate the state of the pointer buttons and modifier keys just prior to the event. The X server can set this member to the bitwise inclusive OR of one or more of the button or modifier key masks: Button1Mask, Button2Mask, Button3Mask, Button4Mask, Button5Mask, ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, Mod5Mask.

The *mode* member is set to indicate whether the events are normal events, pseudo-motion events when a grab activates, or pseudo-motion events when a grab deactivates. The X server can set this member to NotifyNormal, NotifyGrab, or NotifyUngrab.

The *detail* member is set to indicate the notify detail and can be NotifyAncestor, NotifyVirtual, NotifyInferior, NotifyNonlinear, or NotifyNonlinearVirtual.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XDestroyWindowEvent()*, *XErrorEvent()*, *XExposeEvent()*,
XFocusChangeEvent(), *XGraphicsExposeEvent()*, *XGravityEvent()*, *XKeymapEvent()*,
XMapEvent(), *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XDefineCursor(), XUndefineCursor() – define cursors

SYNOPSIS

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

ARGUMENTS

cursor Specifies the cursor that is to be displayed or None.
display Specifies the connection to the X server.
w Specifies the window.

DESCRIPTION

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

XDefineCursor() can generate BadCursor and BadWindow errors.

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

XUndefineCursor() can generate a BadWindow error.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.
BadCursor A value for a Cursor argument does not name a defined Cursor.
BadWindow A value for a Window argument does not name a defined Window.

SEE ALSO

XCreateFontCursor(), *XRecolorCursor()*

Xlib — C Language X Interface

NAME

XDestroyWindow(), XDestroySubwindows() – destroy windows

SYNOPSIS

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

ARGUMENTS

display Specifies the connection to the X server.
w Specifies the window.

DESCRIPTION

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

XDestroyWindow() can generate a BadWindow error.

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

XDestroySubwindows() can generate a BadWindow error.

DIAGNOSTICS

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(), *XConfigureWindow()*, *XCreateWindow()*,
XMapWindow(), *XRaiseWindow()*, *XUnmapWindow()*

Xlib — C Language X Interface

NAME

XDestroyWindowEvent – DestroyNotify event structure

STRUCTURES

The structure for DestroyNotify events contains:

```
typedef struct {
    int type;                DestroyNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window event;
    Window window;
} XDestroyWindowEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *event* member is set either to the destroyed window or to its parent, depending on whether StructureNotify or SubstructureNotify was selected. The *window* member is set to the window that is destroyed.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XErrorEvent()*, *XExposeEvent()*,
XFocusChangeEvent(), *XGraphicsExposeEvent()*, *XGravityEvent()*, *XKeymapEvent()*,
XMapEvent(), *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XDrawArc(), XDrawArcs(), XArc() – draw arcs and arc structure

SYNOPSIS

```
XDrawArc (Display *display, Drawable d, GC gc, int x,
          int y, unsigned int width, height, int angle1,
          int angle2)
XDrawArcs (Display *display, Drawable d, GC gc, XArc *arcs,
           int narcs)
```

ARGUMENTS

angle1 Specifies the start of the arc relative to the three-o'clock position from the center, in units of degrees * 64.

angle2 Specifies the path and extent of the arc relative to the start of the arc, in units of degrees * 64.

arcs Specifies an array of arcs.

d Specifies the drawable.

display Specifies the connection to the X server.

gc Specifies the GC.

narcs Specifies the number of arcs in the array.

width

height Specify the width and height, which are the major and minor axes of the arc.

x

y Specify the x and y coordinates, which are relative to the origin of the drawable and specify the upper-left corner of the bounding rectangle.

DESCRIPTION

XDrawArc() draws a single circular or elliptical arc, and *XDrawArcs()* draws multiple circular or elliptical arcs. Each arc is specified by a rectangle and two angles. The center of the circle or ellipse is the center of the rectangle, and the major and minor axes are specified by the *width* and *height*. Positive angles indicate counterclockwise motion, and negative angles indicate clockwise motion. If the magnitude of *angle2* is greater than 360 degrees, *XDrawArc()* or *XDrawArcs()* truncates it to 360 degrees.

For an arc specified as [*x*, *y*, *width*, *height*, *angle 1*, *angle 2*], the origin of the major and minor axes is at $[x + \frac{width}{2}, y + \frac{height}{2}]$, and the infinitely thin path describing the entire circle or ellipse intersects the horizontal axis at $[x, y + \frac{height}{2}]$ and $[x + width, y + \frac{height}{2}]$ and intersects the vertical axis at $[x + \frac{width}{2}, y]$ and $[x + \frac{width}{2}, y + height]$. These coordinates can be fractional and so are not truncated to discrete coordinates. The path should be defined by the ideal mathematical path. For a wide line with line-width *lw*, the bounding outlines for filling are given by the two infinitely thin paths consisting of all points whose perpendicular distance from the path of the circle/ellipse is equal to *lw* / 2 (which may be a fractional value). The cap-style and join-style are applied the same as for a line corresponding to the tangent of the circle/ellipse at the endpoint.

For an arc specified as [*x*, *y*, *width*, *height*, *angle 1*, *angle 2*], the angles must be specified in the effectively skewed coordinate system of the ellipse (for a circle, the angles and coordinate systems are identical). The relationship between these angles and angles expressed in the normal coordinate system of the screen (as measured with a protractor) is as follows:

$$\text{skewed-angle} = \text{atan}\left(\tan(\text{normal-angle}) * \frac{\text{width}}{\text{height}}\right) + \text{adjust}$$

The skewed-angle and normal-angle are expressed in radians (rather than in degrees scaled by 64) in the range $[0, 2\pi]$ and where atan returns a value in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$ and adjust is:

0	for normal-angle in the range $[0, \frac{\pi}{2}]$
π	for normal-angle in the range $[\frac{\pi}{2}, \frac{3\pi}{2}]$
2π	for normal-angle in the range $[\frac{3\pi}{2}, 2\pi]$

For any given arc, *XDrawArc()* and *XDrawArcs()* do not draw a pixel more than once. If two arcs join correctly and if the line-width is greater than zero and the arcs intersect, *XDrawArc()* and *XDrawArcs()* do not draw a pixel more than once. Otherwise, the intersecting pixels of intersecting arcs are drawn multiple times. Specifying an arc with one endpoint and a clockwise extent draws the same pixels as specifying the other endpoint and an equivalent counterclockwise extent, except as it affects joins.

If the last point in one arc coincides with the first point in the following arc, the two arcs will join correctly. If the first point in the first arc coincides with the last point in the last arc, the two arcs will join correctly. By specifying one axis to be zero, a horizontal or vertical line can be drawn. Angles are computed based solely on the coordinate system and ignore the aspect ratio.

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

XDrawArc() and *XDrawArcs()* can generate BadDrawable, BadGC, and BadMatch errors.

STRUCTURES

The **XArc** structure contains:

```
typedef struct {
    short x, y;
    unsigned short width, height;
    short angle1, angle2;           Degrees * 64
} XArc;
```

All x and y members are signed integers. The *width* and *height* members are 16-bit unsigned integers. You should be careful not to generate coordinates and sizes out of the 16-bit ranges, because the protocol only has 16-bit fields for these values.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to

match in some other way required by the request.

SEE ALSO

XDrawLine(), *XDrawPoint()*, *XDrawRectangle()*

Xlib — C Language X Interface

NAME

XDrawImageString(), XDrawImageString16() – draw image text

SYNOPSIS

```
XDrawImageString (Display *display, Drawable d, GC gc,
                  int x, int y, char *string, int length)
XDrawImageString16 (Display *display, Drawable d, GC gc,
                    int x, int y, XChar2b *string, int length)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
gc Specifies the GC.
length Specifies the number of characters in the string argument.
string Specifies the character string.
x
y Specify the x and y coordinates, which are relative to the origin of the specified drawable and define the origin of the first character.

DESCRIPTION

The *XDrawImageString16()* function is similar to *XDrawImageString()* except that it uses 2-byte or 16-bit characters. Both functions also use both the foreground and background pixels of the GC in the destination.

The effect is first to fill a destination rectangle with the background pixel defined in the GC and then to paint the text with the foreground pixel. The upper-left corner of the filled rectangle is at:

[*x*, *y* – font-ascent]

The width is:

overall-width

The height is:

font-ascent + font-descent

The overall-width, font-ascent, and font-descent are as would be returned by *XQueryTextExtents()* using *gc* and *string*. The function and fill-style defined in the GC are ignored for these functions. The effective function is *GXcopy()*, and the effective fill-style is *FillSolid()*.

For fonts defined with 2-byte matrix indexing and used with *XDrawImageString()*, each byte is used as a byte2 with a byte1 of zero.

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

XDrawImageString() and *XDrawImageString16()* can generate BadDrawable, BadGC, and BadMatch errors.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

SEE ALSO

XDrawString(), *XDrawText()*, *XLoadFont()*, *XTextExtents()*

Xlib — C Language X Interface

NAME

XDrawLine(), XDrawLines(), XDrawSegments(), XSegment() – draw lines, polygons, and line structure

SYNOPSIS

```
XDrawLine (Display *display, Drawable d, GC gc, int x1,
           int y1, int x2, int y2)
XDrawLines (Display *display, Drawable d, GC gc,
            XPoint *points, int npoints, int mode)
XDrawSegments (Display *display, Drawable d, GC gc,
               XSegment *segments, int nsegments)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
gc Specifies the GC.
mode Specifies the coordinate mode. You can pass CoordModeOrigin or CoordModePrevious .
npoints Specifies the number of points in the array.
nsegments Specifies the number of segments in the array.
points Specifies an array of points.
segments Specifies an array of segments.
x1
y1
x2
y2 Specify the points (x1, y1) and (x2, y2) to be connected.

DESCRIPTION

The *XDrawLine()* function uses the components of the specified GC to draw a line between the specified set of points (x1, y1) and (x2, y2). It does not perform joining at coincident endpoints. For any given line, *XDrawLine()* does not draw a pixel more than once. If lines intersect, the intersecting pixels are drawn multiple times.

The *XDrawLines()* function uses the components of the specified GC to draw *npoints*–1 lines between each pair of points (point[i], point[i+1]) in the array of **XPoint** structures. It draws the lines in the order listed in the array. The lines join correctly at all intermediate points, and if the first and last points coincide, the first and last lines also join correctly. For any given line, *XDrawLines()* does not draw a pixel more than once. If thin (zero line-width) lines intersect, the intersecting pixels are drawn multiple times. If wide lines intersect, the intersecting pixels are drawn only once, as though the entire PolyLine protocol request were a single, filled shape. CoordModeOrigin treats all coordinates as relative to the origin, and CoordModePrevious treats all coordinates after the first as relative to the previous point.

The *XDrawSegments()* function draws multiple, unconnected lines. For each segment, *XDrawSegments()* draws a line between (x1, y1) and (x2, y2). It draws the lines in the order listed in the array of **XSegment** structures and does not perform joining at coincident endpoints. For any given line, **XDrawSegments** does not draw a pixel more than once. If lines intersect, the intersecting pixels are drawn multiple times.

All three functions use these GC components: function, plane-mask, line-width, line-style, cap-style, fill-style, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask. The *XDrawLines()* function also uses the join-style GC component. All three functions also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, tile-stipple-y-origin, dash-offset, and dash-list.

XDrawLine(), *XDrawLines()*, and *XDrawSegments()* can generate *BadDrawable*, *BadGC*, and *BadMatch* errors. *XDrawLines()* can also generate a *BadValue* error.

STRUCTURES

The **XSegment** structure contains:

```
typedef struct {
    short x1, y1, x2, y2;
} XSegment;
```

All *x* and *y* members are signed integers. The *width* and *height* members are 16-bit unsigned integers. You should be careful not to generate coordinates and sizes out of the 16-bit ranges, because the protocol only has 16-bit fields for these values.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An **InputOnly** window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XDrawArc(), *XDrawPoint()*, *XDrawRectangle()*

Xlib — C Language X Interface

NAME

XDrawPoint(), XDrawPoints(), XPoint() – draw points and points structure

SYNOPSIS

```
XDrawPoint (Display *display, Drawable d, GC gc, int x,
            int y)
XDrawPoints (Display *display, Drawable d, GC gc,
             XPoint *points, int npoints, int mode)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
gc Specifies the GC.
mode Specifies the coordinate mode. You can pass CoordModeOrigin or CoordModePrevious.
npoints Specifies the number of points in the array.
points Specifies an array of points.
x
y Specify the x and y coordinates where you want the point drawn.

DESCRIPTION

The *XDrawPoint()* function uses the foreground pixel and function components of the GC to draw a single point into the specified drawable; *XDrawPoints()* draws multiple points this way. *CoordModeOrigin()* treats all coordinates as relative to the origin, and *CoordModePrevious()* treats all coordinates after the first as relative to the previous point. *XDrawPoints()* draws the points in the order listed in the array.

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

XDrawPoint() can generate BadDrawable, BadGC, and BadMatch errors. *XDrawPoints()* can generate BadDrawable, BadGC, BadMatch, and BadValue errors.

STRUCTURES

The **XPoint** structure contains:

```
typedef struct {
    short x, y;
} XPoint;
```

All *x* and *y* members are signed integers. The *width* and *height* members are 16-bit unsigned integers. You should be careful not to generate coordinates and sizes out of the 16-bit ranges, because the protocol only has 16-bit fields for these values.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XDrawArc(), *XDrawLine()*, *XDrawRectangle()*

Xlib — C Language X Interface

NAME

XDrawRectangle(), XDrawRectangles(), XRectangle() – draw rectangles and rectangles structure

SYNOPSIS

```
XDrawRectangle (Display *display, Drawable d, GC gc,
                int x, int y, unsigned int width,
                unsigned int height)
XDrawRectangles (Display *display, Drawable d, GC gc,
                 XRectangle rectangles[ ], int nrectangles)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
gc Specifies the GC.
nrectangles Specifies the number of rectangles in the array.
rectangles Specifies an array of rectangles.
width
height Specify the width and height, which specify the dimensions of the rectangle.
x
y Specify the x and y coordinates, which specify the upper-left corner of the rectangle.

DESCRIPTION

The *XDrawRectangle()* and *XDrawRectangles()* functions draw the outlines of the specified rectangle or rectangles as if a five-point PolyLine protocol request were specified for each rectangle:

```
[x,y] [x+width,y] [x+width,y+height] [x,y+height] [x,y]
```

For the specified rectangle or rectangles, these functions do not draw a pixel more than once. *XDrawRectangles()* draws the rectangles in the order listed in the array. If rectangles intersect, the intersecting pixels are drawn multiple times.

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

XDrawRectangle() and *XDrawRectangles()* can generate BadDrawable, BadGC, and BadMatch errors.

STRUCTURES

The **XRectangle** structure contains:

```
typedef struct {
    short x, y;
    unsigned short width, height;
} XRectangle;
```

All x and y members are signed integers. The width and height members are 16-bit unsigned integers. You should be careful not to generate coordinates and sizes out of

the 16-bit ranges, because the protocol only has 16-bit fields for these values.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

SEE ALSO

XDrawArc(), *XDrawLine()*, *XDrawPoint()*

Xlib — C Language X Interface

NAME

XDrawString(), XDrawString16() – draw text characters

SYNOPSIS

```
XDrawString (Display *display, Drawable d, GC gc, int x,
             int y, char *string, int length)
XDrawString16 (Display *display, Drawable d, GC gc, int x,
              int y, XChar2b *string, int length)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
gc Specifies the GC.
length Specifies the number of characters in the *string* argument.
string Specifies the character string.
x
y Specify the x and y coordinates, which are relative to the origin of the specified drawable and define the origin of the first character.

DESCRIPTION

Each character image, as defined by the font in the GC, is treated as an additional mask for a fill operation on the drawable. The drawable is modified only where the font character has a bit set to 1. For fonts defined with 2-byte matrix indexing and used with *XDrawString16()*, each byte is used as a byte2 with a byte1 of zero.

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

XDrawString() and *XDrawString16()* can generate BadDrawable, BadGC, and BadMatch errors.

DIAGNOSTICS

BadDrawable A value for a Drawable argument does not name a defined Window or Pixmap.
 BadGC A value for a GContext argument does not name a defined GContext.
 BadMatch An InputOnly window is used as a Drawable.
 BadMatch Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

SEE ALSO

XDrawImageString(), *XDrawText()*, *XLoadFont()*

Xlib — C Language X Interface

NAME

XDrawText(), XDrawText16(), XTextItem(), XTextItem16() – draw polytext text and text drawing structures

SYNOPSIS

```
XDrawText (Display *display, Drawable d, GC gc, int x,
           int y, XTextItem *items, int nitems)
XDrawText16 (Display *display, Drawable d, GC gc, int x,
             int y, XTextItem16 *items, int nitems)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
gc Specifies the GC.
items Specifies an array of text items.
nitems Specifies the number of text items in the array.
x
y Specify the x and y coordinates, which are relative to the origin of the specified drawable and define the origin of the first character.

DESCRIPTION

The *XDrawText16()* function is similar to *XDrawText()* except that it uses 2-byte or 16-bit characters. Both functions allow complex spacing and font shifts between counted strings.

Each text item is processed in turn. A font member other than None in an item causes the font to be stored in the GC and used for subsequent text. A text element delta specifies an additional change in the position along the x axis before the string is drawn. The delta is always added to the character origin and is not dependent on any characteristics of the font. Each character image, as defined by the font in the GC, is treated as an additional mask for a fill operation on the drawable. The drawable is modified only where the font character has a bit set to 1. If a text item generates a BadFont error, the previous text items may have been drawn.

For fonts defined with linear indexing rather than 2-byte matrix indexing, each **XChar2b** structure is interpreted as a 16-bit number with byte1 as the most-significant byte.

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

XDrawText() and *XDrawText16()* can generate BadDrawable, BadFont, BadGC, and BadMatch errors.

STRUCTURES

The **XTextItem** and **XTextItem16** structures contain:

```
typedef struct {
    char *chars;           pointer to string
    int nchars;            number of characters
    int delta;             delta between strings
    Font font;             Font to print it in, None don't change
} XTextItem;
```

```
typedef struct {
    XChar2b *chars;           pointer to two-byte characters
    int nchars;               number of characters
    int delta;                delta between strings
    Font font;                font to print it in, None don't change
} XTextItem16;
```

If the font member is not None, the font is changed before printing and also is stored in the GC. If an error was generated during text drawing, the previous items may have been drawn. The baseline of the characters are drawn starting at the x and y coordinates that you pass in the text drawing functions.

For example, consider the background rectangle drawn by *XDrawImageString()*. If you want the upper-left corner of the background rectangle to be at pixel coordinate (x,y), pass the (x,y + ascent) as the baseline origin coordinates to the text functions. The ascent is the font ascent, as given in the **XFontStruct** structure. If you want the lower-left corner of the background rectangle to be at pixel coordinate (x,y), pass the (x,y – descent + 1) as the baseline origin coordinates to the text functions. The descent is the font descent, as given in the **XFontStruct** structure.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadFont

A value for a Font or GContext argument does not name a defined Font.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

SEE ALSO

XDrawImageString(), *XDrawString()*, *XLoadFont()*

Xlib — C Language X Interface

NAME

XEmptyRegion(), XEqualRegion(), XPointInRegion(), XRectInRegion() – determine if regions are empty or equal

SYNOPSIS

```

Bool XEmptyRegion (Region r)
Bool XEqualRegion (Region r1, r2)
Bool XPointInRegion (Region r, int x, y)
int XRectInRegion (Region r, int x, int y,
                  unsigned int width, unsigned int height)

```

ARGUMENTS

r Specifies the region.

r1

r2 Specify the two regions.

width

height Specify the width and height, which define the rectangle.

x

y Specify the x and y coordinates, which define the point or the coordinates of the upper-left corner of the rectangle.

DESCRIPTION

The *XEmptyRegion()* function returns True if the region is empty.

The *XEqualRegion()* function returns True if the two regions have the same offset, size, and shape.

The *XPointInRegion()* function returns True if the point (*x*, *y*) is contained in the region *r*.

The *XRectInRegion()* function returns RectangleIn if the rectangle is entirely in the specified region, RectangleOut if the rectangle is entirely out of the specified region, and RectanglePart if the rectangle is partially in the specified region.

SEE ALSO

XCreateRegion(), *XIntersectRegion()*

Xlib — C Language X Interface

NAME

XErrorEvent – X error event structure

STRUCTURES

The **XErrorEvent** structure contains:

```
typedef struct {
    int type;
    Display *display;      Display the event was read from
    unsigned long serial;   serial number of failed request
    unsigned char error_code; error code of failed request
    unsigned char request_code; Major op-code of failed request
    unsigned char minor_code; Minor op-code of failed request
    XID resourceid;        resource id
} XErrorEvent;
```

When you receive this event, the structure members are set as follows.

The *serial* member is the number of requests, starting from one, sent over the network connection since it was opened. It is the number that was the value of NextRequest immediately before the failing call was made. The *request_code* member is a protocol request of the procedure that failed, as defined in <X11/Xproto.h>.

SEE ALSO

AllPlanes(), *XAnyEvent()*, *XButtonEvent()*, *XCreateWindowEvent()*,
XCirculateEvent(), *XCirculateRequestEvent()*, *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XReparentEvent(), *XResizeRequestEvent()*, *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XExposeEvent – Expose event structure

STRUCTURES

The structure for Expose events contains:

```
typedef struct {
    int type;                Expose
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;
    int x, y;
    int width, height;
    int count;               if nonzero, at least this many more
} XExposeEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *window* member is set to the exposed (damaged) window. The *x* and *y* members are set to the coordinates relative to the window's origin and indicate the upper-left corner of the rectangle. The *width* and *height* members are set to the size (extent) of the rectangle. The *count* member is set to the number of Expose events that are to follow. If *count* is zero, no more Expose events follow for this window. However, if *count* is nonzero, at least that number of Expose events (and possibly more) follow for this window. Simple applications that do not want to optimize redisplay by distinguishing between subareas of its window can just ignore all Expose events with nonzero counts and perform full redisplays on events with zero counts.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XFocusChangeEvent(), *XGraphicsExposeEvent()*, *XGravityEvent()*, *XKeymapEvent()*,
XMapEvent(), *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XExtentsOfFontSet() – obtain the maximum extents structure for a font set

SYNOPSIS

```
XFontSetExtents *XExtentsOfFontSet (XFontSet font_set)
```

ARGUMENTS

font_set Specifies the font set.

DESCRIPTION

The *XExtentsOfFontSet()* function returns an **XFontSetExtents** structure for the fonts used by the Xmb and Xwc layers, for the given font set.

The **XFontSetExtents** structure is owned by Xlib and should not be modified or freed by the client. It will be freed by a call to *XFreeFontSet()* with the associated **XFontSet**. Until freed, its contents will not be modified by Xlib.

SEE ALSO

XCreateFontSet(), *XFontsOfFontSet()*, *XFontSetExtents()*

Xlib — C Language X Interface

NAME

XFillRectangle(), XFillRectangles(), XFillPolygon(), XFillArc(), XFillArcs() – fill rectangles, polygons, or arcs

SYNOPSIS

```
XFillRectangle (Display *display, Drawable d, GC gc,
                int x, int y,
                unsigned int width, unsigned int height)
XFillRectangles (Display *display, Drawable d, GC gc,
                 XRectangle *rectangles, int nrectangles)
XFillPolygon (Display *display, Drawable d, GC gc,
              XPoint *points, int npoints, int shape, int mode)
XFillArc (Display *display, Drawable d, GC gc,
          int x, int y, unsigned int width,
          unsigned int height, int angle1, int angle2)
XFillArcs (Display *display, Drawable d, GC gc, XArc *arcs,
           int narcs)
```

ARGUMENTS

angle1 Specifies the start of the arc relative to the three-o'clock position from the center, in units of degrees * 64.

angle2 Specifies the path and extent of the arc relative to the start of the arc, in units of degrees * 64.

arcs Specifies an array of arcs.

d Specifies the drawable.

display Specifies the connection to the X server.

gc Specifies the GC.

mode Specifies the coordinate mode. You can pass CoordModeOrigin or CoordModePrevious.

narcs Specifies the number of arcs in the array.

npoints Specifies the number of points in the array.

nrectangles Specifies the number of rectangles in the array.

points Specifies an array of points.

rectangles Specifies an array of rectangles.

shape Specifies a shape that helps the server to improve performance. You can pass Complex, Convex, or Nonconvex.

width

height Specify the width and height, which are the dimensions of the rectangle to be filled or the major and minor axes of the arc.

x

y Specify the x and y coordinates, which are relative to the origin of the drawable and specify the upper-left corner of the rectangle.

DESCRIPTION

The *XFillRectangle()* and *XFillRectangles()* functions fill the specified rectangle or rectangles as if a four-point FillPolygon protocol request were specified for each rectangle:

```
[x,y] [x+width,y] [x+width,y+height] [x,y+height]
```

Each function uses the *x* and *y* coordinates, *width* and *height* dimensions, and GC you specify.

XFillRectangles() fills the rectangles in the order listed in the array. For any given rectangle, *XFillRectangle()* and *XFillRectangles()* do not draw a pixel more than once. If rectangles intersect, the intersecting pixels are drawn multiple times.

Both functions use these GC components: function, plane-mask, fill-style, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, and tile-stipple-y-origin.

XFillRectangle() and *XFillRectangles()* can generate BadDrawable, BadGC, and BadMatch errors.

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

Depending on the specified shape, the following occurs:

- If shape is *Complex*, the path may self-intersect. 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: function, plane-mask, fill-style, fill-rule, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask. It also uses these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-origin, and tile-stipple-y-origin.

XFillPolygon() can generate BadDrawable, BadGC, BadMatch, and BadValue errors.

For each arc, *XFillArc()* or *XFillArcs()* fills the region closed by the infinitely thin path described by the specified arc and, depending on the arc-mode specified in the GC, one or two line segments. For *ArcChord*, the single line segment joining the endpoints of the arc is used. For *ArcPieSlice*, the two line segments joining the endpoints of the arc with the center point are used. *XFillArcs* fills the arcs in the order listed in the array. For any given arc, *XFillArc()* and *XFillArcs()* do not draw a pixel more than once. If regions intersect, the intersecting pixels are drawn multiple times.

Both functions use these GC components: function, plane-mask, fill-style, arc-mode, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask. They also use these GC mode-dependent components: foreground, background, tile, stipple, tile-stipple-x-

origin, and tile-stipple-y-origin.

XFillArc() and *XFillArcs()* can generate BadDrawable, BadGC, and BadMatch errors.

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XDrawArc(), *XDrawPoint()*, *XDrawRectangle()*

Xlib — C Language X Interface

NAME

XFilterEvent() – filter X events for an input method

SYNOPSIS

Bool XFilterEvent (XEvent **event*, Window *w*)

ARGUMENTS

event Specifies the event to filter.
w Specifies the window for which the filter is to be applied.

DESCRIPTION

If the window argument is None, XFilterEvent applies the filter to the window specified in the **XEvent** structure. The window argument is provided so that layers above Xlib that do event redirection can indicate to which window an event has been redirected.

If *XFilterEvent()* returns True, then some input method has filtered the event, and the client should discard the event. If *XFilterEvent()* returns False, then the client should continue processing the event.

If a grab has occurred in the client, and *XFilterEvent()* returns True, the client should ungrab the keyboard.

SEE ALSO

XNextEvent()

Xlib — C Language X Interface

NAME

XFlush(), XSync(), XEventsQueued(), XPending() – handle output buffer or event queue

SYNOPSIS

```
XFlush (Display *display)
XSync (Display *display, Bool discard)
int XEventsQueued (Display *display, int mode)
int XPending (Display *display)
```

ARGUMENTS

discard Specifies a Boolean value that indicates whether *XSync()* discards all events on the event queue.

display Specifies the connection to the X server.

mode Specifies the mode. You can pass *QueuedAlready*, *QueuedAfterFlush*, or *QueuedAfterReading*.

DESCRIPTION

The *XFlush()* function flushes the output buffer. Most client applications need not use this function because the output buffer is automatically flushed as needed by calls to *XPending()*, *XNextEvent()*, and *XWindowEvent()*. Events generated by the server may be enqueued into the library's event queue.

The *XSync()* function flushes the output buffer and then waits until all requests have been received and processed by the X server. Any errors generated must be handled by the error handler. For each protocol error received by Xlib, *XSync()* calls the client application's error handling routine (see section 11.8.2). Any events generated by the server are enqueued into the library's event queue.

Finally, if you passed *False*, *XSync()* does not discard the events in the queue. If you passed *True*, *XSync()* discards all events in the queue, including those events that were on the queue before *XSync()* was called. Client applications seldom need to call *XSync()*.

If *mode* is *QueuedAlready*, *XEventsQueued()* returns the number of events already in the event queue (and never performs a system call). If *mode* is *QueuedAfterFlush*, *XEventsQueued()* returns the number of events already in the queue if the number is nonzero. If there are no events in the queue, *XEventsQueued()* flushes the output buffer, attempts to read more events out of the application's connection, and returns the number read. If *mode* is *QueuedAfterReading*, *XEventsQueued()* returns the number of events already in the queue if the number is nonzero. If there are no events in the queue, *XEventsQueued()* attempts to read more events out of the application's connection without flushing the output buffer and returns the number read.

XEventsQueued() always returns immediately without I/O if there are events already in the queue. *XEventsQueued()* with *mode* *QueuedAfterFlush* is identical in behavior to *XPending()*. *XEventsQueued()* with *mode* *QueuedAlready* is identical to the *XQLength()* function.

The *XPending()* function returns the number of events that have been received from the X server but have not been removed from the event queue. *XPending()* is identical to *XEventsQueued()* with the *mode* *QueuedAfterFlush* specified.

SEE ALSO

AllPlanes(), *XIfEvent()*, *XNextEvent()*, *XPutBackEvent()*

Xlib — C Language X Interface

NAME

XFocusChangeEvent – FocusIn and FocusOut event structure

STRUCTURES

The structure for FocusIn and FocusOut events contains:

```
typedef struct {
    int type;                FocusIn or FocusOut
    unsigned long serial;    # of last request processed by server
    Bool send_event;        true if this came from a SendEvent request
    Display *display;       Display the event was read from
    Window window;         window of event
    int mode;               NotifyNormal, NotifyGrab, NotifyUngrab
    int detail;             NotifyAncestor, NotifyVirtual, NotifyInferior,
                           NotifyNonlinear, NotifyNonlinearVirtual, NotifyPointer,
                           NotifyPointerRoot, NotifyDetailNone
} XFocusChangeEvent;
typedef XFocusChangeEvent XFocusInEvent;
typedef XFocusChangeEvent XFocusOutEvent;
```

When you receive these events, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to **GraphicsExpose**. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *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.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XGraphicsExposeEvent()*, *XGravityEvent()*, *XKeymapEvent()*,
XMapEvent(), *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,

XSelectionRequestEvent(), XUnmapEvent(), XVisibilityEvent()

Xlib — C Language X Interface

NAME

XFontSetExtents – XFontSetExtents structure

STRUCTURES

The **XFontSetExtents** structure contains:

```
typedef struct {
    XRectangle max_ink_extent;      over all drawable characters
    XRectangle max_logical_extent; over all drawable characters
} XFontSetExtents;
```

The **XRectangles** used to return font set metrics are the usual Xlib screen-oriented **XRectangles**, with *x*, *y* giving the upper left corner, and *width* and *height* always positive.

The *max_ink_extent* member gives the maximum extent, over all drawable characters, of the rectangles which bound the character glyph image drawn in the foreground color, relative to a constant origin. See *XmbTextExtents()* and *XwcTextExtents()* for detailed semantics.

The *max_logical_extent* member gives the maximum extent, over all drawable characters, of the rectangles which specify minimum spacing to other graphical features, relative to a constant origin. Other graphical features drawn by the client, for example, a border surrounding the text, should not intersect this rectangle. The *max_logical_extent* member should be used to compute minimum inter-line spacing and the minimum area which must be allowed in a text field to draw a given number of arbitrary characters.

Due to context-dependent rendering, appending a given character to a string may increase the string's extent by an amount which exceeds the font's max extent:

```
max possible added extent =
    (max_extent * <total # chars>) - prev_string_extent
```

SEE ALSO

XCreateFontSet(), *XExtentsOfFontSet()*, *XFontsOfFontSet()*

Xlib — C Language X Interface

NAME

XFontsOfFontSet(), XBaseFontNameListOfFontSet(), XLocaleOfFontSet(),
XContextDependentDrawing() – obtain fontset information

SYNOPSIS

```
int XFontsOfFontSet (XFontSet font_set,
                    XFontStruct ***font_struct_list_return,
                    char ***font_name_list_return)
char *XBaseFontNameListOfFontSet (XFontSet font_set)
char *XLocaleOfFontSet (XFontSet font_set)
Bool XContextDependentDrawing (XFontSet font_set)
```

ARGUMENTS

font_set Specifies the font set.
font_name_list_return
 Returns the list of font names.
font_struct_list_return
 Returns the list of font structs.

DESCRIPTION

The *XFontsOfFontSet()* function returns a list of one or more **XFontStruct**s and font names for the fonts used by the Xmb and Xwc layers, for the given font set. A list of pointers to the **XFontStruct** structures is returned to *font_struct_list_return*. A list of pointers to null-terminated fully specified font name strings in the locale of the font set is returned to *font_name_list_return*. The *font_name_list* order corresponds to the *font_struct_list* order. The number of **XFontStruct** structures and font names is returned as the value of the function.

Because it is not guaranteed that a given character will be imaged using a single font glyph, there is no provision for mapping a character or default string to the font properties, font ID, or direction hint for the font for the character. The client may access the **XFontStruct** list to obtain these values for all the fonts currently in use.

It is not required that fonts be loaded from the server at the creation of an **XFontSet**. Xlib may choose to cache font data, loading it only as needed to draw text or compute text dimensions. Therefore, existence of the *per_char* metrics in the **XFontStruct** structures in the **XFontStructSet** is undefined. Also, note that all properties in the **XFontStruct** structures are in the STRING encoding.

The **XFontStruct** and font name lists are owned by Xlib and should not be modified or freed by the client. They will be freed by a call to **XFreeFontSet** with the associated **XFontSet**. Until freed, its contents will not be modified by Xlib.

The *XBaseFontNameListOfFontSet()* function returns the original base font name list supplied by the client when the **XFontSet** was created. A null-terminated string containing a list of comma-separated font names is returned as the value of the function. Whitespace may appear immediately on either side of separating commas.

If *XCreateFontSet()* obtained an XLFD name from the font properties for the font specified by a non-XLFD base name, the *XBaseFontNameListOfFontSet()* function will return the XLFD name instead of the non-XLFD base name.

The base font name list is owned by Xlib and should not be modified or freed by the client. It will be freed by a call to *XFreeFontSet()* with the associated **XFontSet**. Until

freed, its contents will not be modified by Xlib.

The *XLocaleOfFontSet()* function returns the name of the locale bound to the specified **XFontSet**, as a null-terminated string.

The returned locale name string is owned by Xlib and should not be modified or freed by the client. It may be freed by a call to *XFreeFontSet()* with the associated *XFontSet()*. Until freed, it will not be modified by Xlib.

The *XContextDependentDrawing()* function returns True if text drawn with the *font_set* might include context-dependent drawing.

SEE ALSO

XCreateFontSet(), *XExtentsOfFontSet()*, *XFontSetExtents()*

Xlib — C Language X Interface

NAME

XFree() – free client data

SYNOPSIS

XFree (void **data*)

ARGUMENTS

data Specifies the data that is to be freed.

DESCRIPTION

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

SEE ALSO

Xlib — C Language X Interface

NAME

XGetVisualInfo(), XMatchVisualInfo(), XVisualIDFromVisual(), XVisualInfo() – obtain visual information and visual structure

SYNOPSIS

```
XVisualInfo *XGetVisualInfo (Display *display, long vinfo_mask,
                             XVisualInfo *vinfo_template,
                             int *nitems_return)
Status XMatchVisualInfo (Display *display, int screen, int depth,
                        int class, XVisualInfo *vinfo_return)
VisualID XVisualIDFromVisual (Visual * visual)
```

ARGUMENTS

class Specifies the class of the screen.
depth Specifies the depth of the screen.
display Specifies the connection to the X server.
nitems_return
 Returns the number of matching visual structures.
screen Specifies the screen.
visual Specifies the visual type.
vinfo_mask
 Specifies the visual mask value.
vinfo_return
 Returns the matched visual information.
vinfo_template
 Specifies the visual attributes that are to be used in matching the visual structures.

DESCRIPTION

The *XGetVisualInfo()* function returns a list of visual structures that have attributes equal to the attributes specified by *vinfo_template*. If no visual structures match the template using the specified *vinfo_mask*, *XGetVisualInfo()* returns a NULL. To free the data returned by this function, use *XFree()*.

The *XMatchVisualInfo()* function returns the visual information for a visual that matches the specified depth and class for a screen. Because multiple visuals that match the specified depth and class can exist, the exact visual chosen is undefined. If a visual is found, *XMatchVisualInfo()* returns nonzero and the information on the visual to *vinfo_return*. Otherwise, when a visual is not found, *XMatchVisualInfo()* returns zero.

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

STRUCTURES

The **XVisualInfo** structure contains:

Visual information mask bits

```
#define VisualNoMask          0x0
#define VisualIDMask          0x1
```

```
#define VisualScreenMask      0x2
#define VisualDepthMask      0x4
#define VisualClassMask      0x8
#define VisualRedMaskMask    0x10
#define VisualGreenMaskMask  0x20
#define VisualBlueMaskMask   0x40
#define VisualColormapSizeMask 0x80
#define VisualBitsPerRGBMask 0x100
#define VisualAllMask        0x1FF
```

Values

```
typedef struct {
    Visual *visual;
    VisualID visualid;
    int screen;
    unsigned int depth;
    int class;
    unsigned long red_mask;
    unsigned long green_mask;
    unsigned long blue_mask;
    int colormap_size;
    int bits_per_rgb;
} XVisualInfo;
```

SEE ALSO*XFree()**Xlib — C Language X Interface*

NAME

XGetWindowAttributes(), XGetGeometry(), XWindowAttributes() – get current window attribute or geometry and current window attributes structure

SYNOPSIS

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

ARGUMENTS

border_width_return
Returns the border width in pixels.

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

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

display
Specifies the connection to the X server.

root_return
Returns the root window.

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

width_return
height_return
Return the drawable's dimensions (width and height).

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

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

DESCRIPTION

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

XGetWindowAttributes() can generate BadDrawable and BadWindow errors.

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

STRUCTURES

The **XWindowAttributes** structure contains:

```
typedef struct {
    int x, y;                location of window
    int width, height;       width and height of window
```


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

`} XWindowAttributes;`

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

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

<code>ForgetGravity</code>	<code>EastGravity</code>
<code>NorthWestGravity</code>	<code>SouthWestGravity</code>
<code>NorthGravity</code>	<code>SouthGravity</code>
<code>NorthEastGravity</code>	<code>SouthEastGravity</code>
<code>WestGravity</code>	<code>StaticGravity</code>
<code>CenterGravity</code>	

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

<code>UnmapGravity</code>	<code>EastGravity</code>
<code>NorthWestGravity</code>	<code>SouthWestGravity</code>
<code>NorthEastGravity</code>	<code>SouthEastGravity</code>
<code>WestGravity</code>	<code>StaticGravity</code>
<code>CenterGravity</code>	

For additional information on gravity, see section 3.3.

The `backing_store` member is set to indicate how the X server should maintain the contents of a window and can be `WhenMapped`, `Always`, or `NotUseful`. The `backing_planes` member is set to indicate (with bits set to 1) which bit planes of the

window hold dynamic data that must be preserved in *backing_stores* and during *save_unders*. The *backing_pixel* member is set to indicate what values to use for planes not set in *backing_planes*.

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

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

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

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

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XQueryPointer(), *XQueryTree()*

Xlib — C Language X Interface

NAME

XGetWindowProperty(), XListProperties(), XChangeProperty(),
XRotateWindowProperties(), XDeleteProperty() – obtain and change window properties

SYNOPSIS

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

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

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

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

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

ARGUMENTS

actual_format_return
Returns the actual format of the property.

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

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

data
Specifies the property data.

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

display
Specifies the connection to the X server.

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

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

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

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

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

<i>nitems_return</i>	Returns the actual number of 8-bit, 16-bit, or 32-bit items stored in the <i>prop_return</i> data.
<i>num_prop</i>	Specifies the length of the <i>properties</i> array.
<i>num_prop_return</i>	Returns the length of the properties array.
<i>npositions</i>	Specifies the rotation amount.
<i>prop_return</i>	Returns the data in the specified format.
<i>property</i>	Specifies the property name.
<i>properties</i>	Specifies the array of properties that are to be rotated.
<i>req_type</i>	Specifies the atom identifier associated with the property type or AnyPropertyType .
<i>type</i>	Specifies the type of the property. The X server does not interpret the type but simply passes it back to an application that later calls XGetWindowProperty .
<i>w</i>	Specifies the window whose property you want to obtain, change, rotate or delete.

DESCRIPTION

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

- If the specified property does not exist for the specified window, *XGetWindowProperty()* returns None to *actual_type_return* and the value zero to *actual_format_return* and *bytes_after_return*. The *nitems_return* argument is empty. In this case, the *delete* argument is ignored.
- If the specified property exists but its type does not match the specified type, *XGetWindowProperty()* returns the actual property type to *actual_type_return*, the actual property format (never zero) to *actual_format_return*, and the property length in bytes (even if the *actual_format_return* is 16 or 32) to *bytes_after_return*. It also ignores the *delete* argument. The *nitems_return* argument is empty.
- If the specified property exists and either you assign AnyPropertyType to the *req_type* argument or the specified type matches the actual property type, *XGetWindowProperty()* returns the actual property type to *actual_type_return* and the actual property format (never zero) to *actual_format_return*. It also returns a value to *bytes_after_return* and *nitems_return*, by defining the following values:

N	actual length of the stored property in bytes (even if the format is 16 or 32)
I	4 * long_offset
T	N - I
L	MINIMUM(T, 4 * long_length)
A	N - (I + L)
- The returned value starts at byte index I in the property (indexing from zero), and its length in bytes is L. If the value for *long_offset* causes L to be negative, a BadValue error results. The value of *bytes_after_return* is A, giving the number of trailing unread bytes in the stored property.

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

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

XGetWindowProperty() can generate BadAtom, BadValue, and BadWindow errors.

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

XListProperties() can generate a BadWindow error.

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

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

The lifetime of a property is not tied to the storing client. Properties remain until explicitly deleted, until the window is destroyed, or until the server resets. For a discussion of what happens when the connection to the X server is closed, see section 2.6. The maximum size of a property is server dependent and can vary dynamically depending on the amount of memory the server has available. (If there is insufficient space, a BadAlloc error results.)

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

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

XRotateWindowProperties() can generate BadAtom, BadMatch, and BadWindow errors.

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

XDeleteProperty() can generate BadAtom and BadWindow errors.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadAtom

A value for an Atom argument does not name a defined Atom.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XFree(), *XInternAtom()*

Xlib — C Language X Interface

NAME

XGrabButton(), XUngrabButton() – grab pointer buttons

SYNOPSIS

```
XGrabButton (Display *display, unsigned int button,
             unsigned int modifiers, Window grab_window,
             Bool owner_events, unsigned int event_mask,
             int pointer_mode, keyboard_mode,
             Window confine_to, Cursor cursor)
XUngrabButton (Display *display, unsigned int button,
              unsigned int modifiers, Window grab_window)
```

ARGUMENTS

button Specifies the pointer button that is to be grabbed or released or AnyButton.

confine_to Specifies the window to confine the pointer in or None .

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

display Specifies the connection to the X server.

event_mask Specifies which pointer events are reported to the client. The mask is the bitwise inclusive OR of the valid pointer event mask bits.

grab_window Specifies the grab window.

keyboard_mode Specifies further processing of keyboard events. You can pass GrabModeSync or GrabModeAsync.

modifiers Specifies the set of keymasks or AnyModifier. The mask is the bitwise inclusive OR of the valid keymask bits.

owner_events Specifies a Boolean value that indicates whether the pointer events are to be reported as usual or reported with respect to the grab window if selected by the event mask.

pointer_mode Specifies further processing of pointer events. You can pass GrabModeSync or GrabModeAsync.

DESCRIPTION

The *XGrabButton()* function establishes a passive grab. In the future, the pointer is actively grabbed (as for *XGrabPointer()*), the last-pointer-grab time is set to the time at which the button was pressed (as transmitted in the ButtonPress event), and the *ButtonPress()* event is reported if all of the following conditions are true:

- The pointer is not grabbed, and the specified button is logically pressed when the specified modifier keys are logically down, and no other buttons or modifier keys are logically down.
- The *grab_window* contains the pointer.
- The *confine_to* window (if any) is viewable.
- A passive grab on the same button / key combination does not exist on any ancestor of *grab_window*.

The interpretation of the remaining arguments is as for *XGrabPointer()*. The active grab is terminated automatically when the logical state of the pointer has all buttons released (independent of the state of the logical modifier keys).

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

This request overrides all previous grabs by the same client on the same button / key combinations on the same window. A modifiers of *AnyModifier* is equivalent to issuing the grab request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned KeyCodes. A button of *AnyButton* is equivalent to issuing the request for all possible buttons. Otherwise, it is not required that the specified button currently be assigned to a physical button.

If some other client has already issued a *XGrabButton()* with the same button / key combination on the same window, a *BadAccess* error results. When using *AnyModifier* or *AnyButton*, the request fails completely, and a *BadAccess* error results (no grabs are established) if there is a conflicting grab for any combination. *XGrabButton()* has no effect on an active grab.

XGrabButton() can generate *BadCursor*, *BadValue*, and *BadWindow* errors.

The *XUngrabButton()* function releases the passive button / key combination on the specified window if it was grabbed by this client. A modifiers of *AnyModifier()* is equivalent to issuing the ungrab request for all possible modifier combinations, including the combination of no modifiers. A button of *AnyButton()* is equivalent to issuing the request for all possible buttons. *XUngrabButton()* has no effect on an active grab.

XUngrabButton() can generate *BadValue* and *BadWindow* errors.

DIAGNOSTICS

BadCursor

A value for a Cursor argument does not name a defined Cursor.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowEvents(), *XGrabPointer()*, *XGrabKey()*, *XGrabKeyboard()*,

Xlib — C Language X Interface

NAME

XGrabKey(), XUngrabKey() – grab keyboard keys

SYNOPSIS

```
XGrabKey (Display *display, int keycode, unsigned int modifiers,
          Window grab_window, Bool owner_events,
          int pointer_mode, keyboard_mode)
XUngrabKey (Display *display, int keycode, unsigned int modifiers,
            Window grab_window)
```

ARGUMENTS

display Specifies the connection to the X server.

grab_window Specifies the grab window.

keyboard_mode Specifies further processing of keyboard events. You can pass GrabModeSync or GrabModeAsync.

keycode Specifies the KeyCode or AnyKey.

modifiers Specifies the set of keymasks or AnyModifier. The mask is the bitwise inclusive OR of the valid keymask bits.

owner_events Specifies a Boolean value that indicates whether the keyboard events are to be reported as usual.

pointer_mode Specifies further processing of pointer events. You can pass GrabModeSync or GrabModeAsync.

DESCRIPTION

The *XGrabKey()* function establishes a passive grab on the keyboard. In the future, the keyboard is actively grabbed (as for *XGrabKeyboard()*), the last-keyboard-grab time is set to the time at which the key was pressed (as transmitted in the KeyPress event), and the KeyPress event is reported if all of the following conditions are true:

- The keyboard is not grabbed and the specified key (which can itself be a modifier key) is logically pressed when the specified modifier keys are logically down, and no other modifier keys are logically down.
- Either the *grab_window* is an ancestor of (or is) the focus window, or the *grab_window* is a descendant of the focus window and contains the pointer.
- A passive grab on the same key combination does not exist on any ancestor of *grab_window*.

The interpretation of the remaining arguments is as for *XGrabKeyboard()*. The active grab is terminated automatically when the logical state of the keyboard has the specified key released (independent of the logical state of the modifier keys).

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

A modifiers argument of AnyModifier is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned KeyCodes. A keycode

argument of *AnyKey* is equivalent to issuing the request for all possible *KeyCodes*. Otherwise, the specified keycode must be in the range specified by *min_keycode* and *max_keycode* in the connection setup, or a *BadValue* error results.

If some other client has issued a *XGrabKey* with the same key combination on the same window, a *BadAccess* error results. When using *AnyModifier* or *AnyKey*, the request fails completely, and a *BadAccess* error results (no grabs are established) if there is a conflicting grab for any combination.

XGrabKey() can generate *BadAccess*, *BadValue*, and *BadWindow* errors.

The *XUngrabKey()* function releases the key combination on the specified window if it was grabbed by this client. It has no effect on an active grab. A modifier of *AnyModifier* is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). A keycode argument of *AnyKey()* is equivalent to issuing the request for all possible key codes.

XUngrabKey() can generate *BadValue* and *BadWindow* error.

DIAGNOSTICS

BadAccess

A client attempted to grab a key/button combination already grabbed by another client.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a *Window* argument does not name a defined *Window*.

SEE ALSO

XGrabButton(), *XGrabKeyboard()*, *XGrabPointer()*

Xlib — C Language X Interface

NAME

XGrabKeyboard(), XUngrabKeyboard() – grab the keyboard

SYNOPSIS

```
int XGrabKeyboard (Display *display, Window grab_window,
                  Bool owner_events, int pointer_mode,
                  int keyboard_mode, Time time)
XUngrabKeyboard (Display *display, Time time)
```

ARGUMENTS

display Specifies the connection to the X server.

grab_window
Specifies the grab window.

keyboard_mode
Specifies further processing of keyboard events. You can pass `GrabModeSync` or `GrabModeAsync`.

owner_events
Specifies a Boolean value that indicates whether the keyboard events are to be reported as usual.

pointer_mode
Specifies further processing of pointer events. You can pass `GrabModeSync` or `GrabModeAsync`.

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

DESCRIPTION

The `XGrabKeyboard()` function actively grabs control of the keyboard and generates `FocusIn` and `FocusOut` events. Further key events are reported only to the grabbing client. `XGrabKeyboard()` overrides any active keyboard grab by this client. If *owner_events* is `False`, all generated key events are reported with respect to *grab_window*. If *owner_events* is `True` and if a generated key event would normally be reported to this client, it is reported normally; otherwise, the event is reported with respect to the *grab_window*. Both `KeyPress` and `KeyRelease` events are always reported, independent of any event selection made by the client.

If the *keyboard_mode* argument is `GrabModeAsync()`, keyboard event processing continues as usual. If the keyboard is currently frozen by this client, then processing of keyboard events is resumed. If the *keyboard_mode* argument is `GrabModeSync()`, the state of the keyboard (as seen by client applications) appears to freeze, and the X server generates no further keyboard events until the grabbing client issues a releasing `XAllowEvents()` call or until the keyboard grab is released. Actual keyboard changes are not lost while the keyboard is frozen; they are simply queued in the server for later processing.

If *pointer_mode* is `GrabModeAsync`, pointer event processing is unaffected by activation of the grab. If *pointer_mode* is `GrabModeSync`, the state of the pointer (as seen by client applications) appears to freeze, and the X server generates no further pointer events until the grabbing client issues a releasing `XAllowEvents()` call or until the keyboard grab is released. Actual pointer changes are not lost while the pointer is frozen; they are simply queued in the server for later processing.

If the keyboard is actively grabbed by some other client, `XGrabKeyboard()` fails and returns `AlreadyGrabbed`. If *grab_window* is not viewable, it fails and returns `GrabNotViewable`. If the keyboard is frozen by an active grab of another client, it fails

and returns `GrabFrozen`. If the specified time is earlier than the last-keyboard-grab time or later than the current X server time, it fails and returns `GrabInvalidTime`. Otherwise, the last-keyboard-grab time is set to the specified time (`CurrentTime` is replaced by the current X server time).

`XGrabKeyboard()` can generate `BadValue` and `BadWindow` errors.

The `XUngrabKeyboard()` function releases the keyboard and any queued events if this client has it actively grabbed from either `XGrabKeyboard()` or `XGrabKey()`.

`XUngrabKeyboard()` does not release the keyboard and any queued events if the specified time is earlier than the last-keyboard-grab time or is later than the current X server time. It also generates `FocusIn` and `FocusOut` events. The X server automatically performs an `UngrabKeyboard()` request if the event window for an active keyboard grab becomes not viewable.

DIAGNOSTICS

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

`XAllowEvents()`, `XGrabButton()`, `XGrabKey()`, `XGrabPointer()`

Xlib — C Language X Interface

NAME

XGrabPointer(), XUngrabPointer(), XChangeActivePointerGrab() – grab the pointer

SYNOPSIS

```
int XGrabPointer (Display *display, Window grab_window,
                  Bool owner_events, unsigned int event_mask,
                  int pointer_mode, keyboard_mode,
                  Window confine_to, Cursor cursor, Time time)
XUngrabPointer (Display *display, Time time)
XChangeActivePointerGrab (Display *display, unsigned int event_mask,
                          Cursor cursor, Time time)
```

ARGUMENTS

confine_to
Specifies the window to confine the pointer in or None.

cursor
Specifies the cursor that is to be displayed during the grab or None.

display
Specifies the connection to the X server.

event_mask
Specifies which pointer events are reported to the client. The mask is the bitwise inclusive OR of the valid pointer event mask bits.

grab_window
Specifies the grab window.

keyboard_mode
Specifies further processing of keyboard events. You can pass GrabModeSync or GrabModeAsync.

owner_events
Specifies a Boolean value that indicates whether the pointer events are to be reported as usual or reported with respect to the grab window if selected by the event mask.

pointer_mode
Specifies further processing of pointer events. You can pass GrabModeSync or GrabModeAsync.

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

DESCRIPTION

The *XGrabPointer()* function actively grabs control of the pointer and returns GrabSuccess if the grab was successful. Further pointer events are reported only to the grabbing client. XGrabPointer overrides any active pointer grab by this client. If *owner_events* is False, all generated pointer events are reported with respect to *grab_window* and are reported only if selected by *event_mask*. If *owner_events* is True and if a generated pointer event would normally be reported to this client, it is reported as usual. Otherwise, the event is reported with respect to the *grab_window* and is reported only if selected by *event_mask*. For either value of *owner_events*, unreported events are discarded.

If the *pointer_mode* is GrabModeAsync, pointer event processing continues as usual. If the pointer is currently frozen by this client, the processing of events for the pointer is resumed. If the *pointer_mode* is GrabModeSync, the state of the pointer, as seen by client applications, appears to freeze, and the X server generates no further pointer events until the grabbing client calls XAllowEvents or until the pointer grab is released. Actual pointer changes are not lost while the pointer is frozen; they are simply queued in the

server for later processing.

If the *keyboard_mode* is *GrabModeAsync*, keyboard event processing is unaffected by activation of the grab. If the *keyboard_mode* is *GrabModeSync*, the state of the keyboard, as seen by client applications, appears to freeze, and the X server generates no further keyboard events until the grabbing client calls *XAllowEvents* or until the pointer grab is released. Actual keyboard changes are not lost while the pointer is frozen; they are simply queued in the server for later processing.

If a cursor is specified, it is displayed regardless of what window the pointer is in. If *None* is specified, the normal cursor for that window is displayed when the pointer is in *grab_window* or one of its subwindows; otherwise, the cursor for *grab_window* is displayed.

If a *confine_to* window is specified, the pointer is restricted to stay contained in that window. The *confine_to* window need have no relationship to the *grab_window*. If the pointer is not initially in the *confine_to* window, it is warped automatically to the closest edge just before the grab activates and enter/leave events are generated as usual. If the *confine_to* window is subsequently reconfigured, the pointer is warped automatically, as necessary, to keep it contained in the window.

The time argument allows you to avoid certain circumstances that come up if applications take a long time to respond or if there are long network delays. Consider a situation where you have two applications, both of which normally grab the pointer when clicked on. If both applications specify the timestamp from the event, the second application may wake up faster and successfully grab the pointer before the first application. The first application then will get an indication that the other application grabbed the pointer before its request was processed.

XGrabPointer generates *EnterNotify* and *LeaveNotify* events.

Either if *grab_window* or *confine_to* window is not viewable or if the *confine_to* window lies outside the root window, *XGrabPointer* fails and returns *GrabNotViewable*. If the pointer is actively grabbed by some other client, it fails and returns *AlreadyGrabbed*. If the pointer is frozen by an active grab of another client, it fails and returns *GrabFrozen*. If the specified time is earlier than the last-pointer-grab time or later than the current X server time, it fails and returns *GrabInvalidTime*. Otherwise, the last-pointer-grab time is set to the specified time (*CurrentTime* is replaced by the current X server time).

XGrabPointer() can generate *BadCursor*, *BadValue*, and *BadWindow* errors.

The *XUngrabPointer()* function releases the pointer and any queued events if this client has actively grabbed the pointer from *XGrabPointer()*, *XGrabButton()*, or from a normal button press. *XUngrabPointer()* does not release the pointer if the specified time is earlier than the last-pointer-grab time or is later than the current X server time. It also generates *EnterNotify* and *LeaveNotify* events. The X server performs an *UngrabPointer()* request automatically if the event window or *confine_to* window for an active pointer grab becomes not viewable or if window reconfiguration causes the *confine_to* window to lie completely outside the boundaries of the root window.

The *XChangeActivePointerGrab()* function changes the specified dynamic parameters if the pointer is actively grabbed by the client and if the specified time is no earlier than the last-pointer-grab time and no later than the current X server time. This function has no effect on the passive parameters of a *XGrabButton()*. The interpretation of *event_mask* and cursor is the same as described in *XGrabPointer*.

XChangeActivePointerGrab() can generate a BadCursor and BadValue error.

DIAGNOSTICS**BadCursor**

A value for a Cursor argument does not name a defined Cursor.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowEvents(), *XGrabButton()*, *XGrabKey()*, *XGrabKeyboard()*

Xlib — C Language X Interface

NAME

XGrabServer(), XUngrabServer() – grab the server

SYNOPSIS

XGrabServer (Display **display*)
XUngrabServer (Display **display*)

ARGUMENTS

display Specifies the connection to the X server.

DESCRIPTION

The *XGrabServer()* function disables processing of requests and close downs on all other connections than the one this request arrived on. You should not grab the X server any more than is absolutely necessary.

The *XUngrabServer()* function restarts processing of requests and close downs on other connections. You should avoid grabbing the X server as much as possible.

SEE ALSO

XGrabButton(), *XGrabKey()*, *XGrabKeyboard()*, *XGrabPointer()*

Xlib — C Language X Interface

NAME

XGraphicsExposeEvent, XNoExposeEvent – GraphicsExpose and NoExpose event structures

STRUCTURES

The structures for GraphicsExpose and NoExpose events contain:

```
typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Drawable drawable;
    int x, y;
    int width, height;
    int count;
    int major_code;
    int minor_code;
} XGraphicsExposeEvent;

typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Drawable drawable;
    int major_code;
    int minor_code;
} XNoExposeEvent;
```

	GraphicsExpose
	# of last request processed by server
	true if this came from a SendEvent request
	Display the event was read from
	if nonzero, at least this many more
	core is CopyArea or CopyPlane
	not defined in the core
	NoExpose
	# of last request processed by server
	true if this came from a SendEvent request
	Display the event was read from
	core is CopyArea or CopyPlane
	not defined in the core

When you receive these events, their structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

Both structures have these common members: *drawable*, *major_code*, and *minor_code*. The *drawable* member is set to the drawable of the destination region on which the graphics request was to be performed. The *major_code* member is set to the graphics request initiated by the client and can be either X_CopyArea or X_CopyPlane. If it is X_CopyArea, a call to *XCopyArea()* initiated the request. If it is X_CopyPlane, a call to *XCopyPlane()* initiated the request. These constants are defined in <X11/Xproto.h>. The *minor_code* member, like the *major_code* member, indicates which graphics request was initiated by the client. However, the *minor_code* member is not defined by the core X protocol and will be zero in these cases, although it may be used by an extension.

The **XGraphicsExposeEvent** structure has these additional members: *x*, *y*, *width*, *height*, and *count*. The *x* and *y* members are set to the coordinates relative to the drawable's

origin and indicate the upper-left corner of the rectangle. The *width* and *height* members are set to the size (extent) of the rectangle. The *count* member is set to the number of GraphicsExpose events to follow. If count is zero, no more GraphicsExpose events follow for this window. However, if count is nonzero, at least that number of GraphicsExpose events (and possibly more) are to follow for this window.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCopyArea()*, *XCrossingEvent()*, *XDestroyWindowEvent()*,
XErrorEvent(), *XExposeEvent()*, *XFocusChangeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XReparentEvent(), *XResizeRequestEvent()*, *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XGravityEvent – GravityNotify event structure

STRUCTURES

The structure for GravityNotify events contains:

```
typedef struct {
    int type;                GravityNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window event;
    Window window;
    int x, y;
} XGravityEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *event* member is set either to the window that was moved or to its parent, depending on whether StructureNotify or SubstructureNotify was selected. The *window* member is set to the child window that was moved. The *x* and *y* members are set to the coordinates relative to the new parent window's origin and indicate the position of the upper-left outside corner of the window.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XKeymapEvent()*,
XMapEvent(), *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XIconifyWindow(), XWithdrawWindow(), XReconfigureWMWindow() – manipulate top-level windows

SYNOPSIS

```
Status XIconifyWindow (Display *display, Window w,
                      int screen_number)
Status XWithdrawWindow (Display *display, Window w,
                      int screen_number)
Status XReconfigureWMWindow (Display *display, Window w,
                          int screen_number,
                          unsigned int value_mask,
                          XWindowChanges *values)
```

ARGUMENTS

display Specifies the connection to the X server.

screen_number Specifies the appropriate screen number on the host server.

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

values Specifies the **XWindowChanges** structure.

w Specifies the window.

DESCRIPTION

The *XIconifyWindow()* function sends a WM_CHANGE_STATE ClientMessage event with a format of 32 and a first data element of IconicState (as described in section 4.1.4 of the *Inter-Client Communication Conventions Manual*) and a window of *w* to the root window of the specified screen with an event mask set to SubstructureNotifyMask | SubstructureRedirectMask. Window managers may elect to receive this message and if the window is in its normal state, may treat it as a request to change the window's state from normal to iconic. If the WM_CHANGE_STATE property cannot be interned, XIconifyWindow does not send a message and returns a zero status. It returns a nonzero status if the client message is sent successfully; otherwise, it returns a zero status.

The *XWithdrawWindow()* function unmaps the specified window and sends a synthetic UnmapNotify event to the root window of the specified screen. Window managers may elect to receive this message and may treat it as a request to change the window's state to withdrawn. When a window is in the withdrawn state, neither its normal nor its iconic representations is visible. It returns a nonzero status if the UnmapNotify event is successfully sent; otherwise, it returns a zero status.

XWithdrawWindow() can generate a BadWindow error.

The *XReconfigureWMWindow()* function issues a ConfigureWindow request on the specified top-level window. If the stacking mode is changed and the request fails with a BadMatch error, the error is trapped by Xlib and a synthetic ConfigureRequestEvent containing the same configuration parameters is sent to the root of the specified window. Window managers may elect to receive this event and treat it as a request to reconfigure the indicated window. It returns a nonzero status if the request or event is

successfully sent; otherwise, it returns a zero status.

XReconfigureWMWindow() can generate *BadValue* and *BadWindow* errors.

DIAGNOSTICS**BadValue**

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(), *XConfigureWindow()*, *XCreateWindow()*,
XDestroyWindow(), *XRaiseWindow()*, *XMapWindow()*, *XUnmapWindow()*

Xlib — C Language X Interface

NAME

XIfEvent(), XCheckIfEvent(), XPeekIfEvent() – check the event queue with a predicate procedure

SYNOPSIS

```
XIfEvent (Display *display, XEvent *event_return,
          Bool (*predicate) (), XPointer arg)
Bool XCheckIfEvent (Display *display, XEvent *event_return,
                   Bool (*predicate) (), XPointer arg)
XPeekIfEvent (Display *display, XEvent *event_return,
              Bool (*predicate) (), XPointer arg)
```

ARGUMENTS

arg Specifies the user-supplied argument that will be passed to the predicate procedure.

display Specifies the connection to the X server.

event_return Returns either a copy of or the matched event's associated structure.

predicate Specifies the procedure that is to be called to determine if the next event in the queue matches what you want.

DESCRIPTION

The *XIfEvent()* function completes only when the specified predicate procedure returns True for an event, which indicates an event in the queue matches. *XIfEvent()* flushes the output buffer if it blocks waiting for additional events. *XIfEvent()* removes the matching event from the queue and copies the structure into the client-supplied **XEvent** structure.

When the predicate procedure finds a match, *XCheckIfEvent()* copies the matched event into the client-supplied **XEvent** structure and returns True. (This event is removed from the queue.) If the predicate procedure finds no match, *XCheckIfEvent()* returns False, and the output buffer will have been flushed. All earlier events stored in the queue are not discarded.

The *XPeekIfEvent()* function returns only when the specified predicate procedure returns True for an event. After the predicate procedure finds a match, *XPeekIfEvent()* copies the matched event into the client-supplied **XEvent** structure without removing the event from the queue. *XPeekIfEvent()* flushes the output buffer if it blocks waiting for additional events.

SEE ALSO

XAnyEvent(), *XNextEvent()*, *XPutBackEvent()* *XSendEvent()*

Xlib — C Language X Interface

NAME

XInstallColormap(), XUninstallColormap(), XListInstalledColormaps() – control colormaps

SYNOPSIS

```
XInstallColormap (Display *display, Colormap colormap)
XUninstallColormap (Display *display, Colormap colormap)
Colormap *XListInstalledColormaps (Display *display, Window w,
                                   int *num_return)
```

ARGUMENTS

colormap Specifies the colormap.

display Specifies the connection to the X server.

num_return Returns the number of currently installed colormaps.

w Specifies the window that determines the screen.

DESCRIPTION

The *XInstallColormap()* function installs the specified colormap for its associated screen. All windows associated with this colormap immediately display with true colors. You associated the windows with this colormap when you created them by calling *XCreateWindow()*, *XCreateSimpleWindow()*, *XChangeWindowAttributes()*, or *XSetWindowColormap()*.

If the specified colormap is not already an installed colormap, the X server generates a ColormapNotify event on each window that has that colormap. In addition, for every other colormap that is installed as a result of a call to *XInstallColormap()*, the X server generates a ColormapNotify event on each window that has that colormap.

XInstallColormap() can generate a BadColor error.

The *XUninstallColormap()* function removes the specified colormap from the required list for its screen. As a result, the specified colormap might be uninstalled, and the X server might implicitly install or uninstall additional colormaps. Which colormaps get installed or uninstalled is server-dependent except that the required list must remain installed.

If the specified colormap becomes uninstalled, the X server generates a ColormapNotify event on each window that has that colormap. In addition, for every other colormap that is installed or uninstalled as a result of a call to *XUninstallColormap()*, the X server generates a ColormapNotify event on each window that has that colormap.

XUninstallColormap() can generate a BadColor error.

The *XListInstalledColormaps()* function returns a list of the currently installed colormaps for the screen of the specified window. The order of the colormaps in the list is not significant and is no explicit indication of the required list. When the allocated list is no longer needed, free it by using *XFree()*.

XListInstalledColormaps() can generate a BadWindow error.

DIAGNOSTICS

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(), *XCreateColormap()*, *XCreateWindow()*, *XFree()*

Xlib — C Language X Interface

NAME

XInternAtom(), XGetAtomName() – create or return atom names

SYNOPSIS

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

ARGUMENTS

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

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

display Specifies the connection to the X server.

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

DESCRIPTION

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

XInternAtom() can generate BadAlloc and BadValue errors.

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

XGetAtomName() can generate a BadAtom error.

DIAGNOSTICS

BadAlloc
The server failed to allocate the requested resource or server memory.

BadAtom
A value for an Atom argument does not name a defined Atom.

BadValue
Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XFree(), *XGetWindowProperty()*

Xlib — C Language X Interface

NAME

XIntersectRegion(), XUnionRegion(), XUnionRectWithRegion(), XSubtractRegion(),
 XXorRegion(), XOffsetRegion(), XShrinkRegion() – region arithmetic

SYNOPSIS

```
XIntersectRegion (Region sra, srb, dr_return)
XUnionRegion (Region sra, srb, dr_return)
XUnionRectWithRegion (XRectangle *rectangle, Region src_region,
                      Region dest_region_return)
XSubtractRegion (Region sra, srb, dr_return)
XXorRegion (Region sra, srb, dr_return)
XOffsetRegion (Region r, int dx, dy)
XShrinkRegion (Region r, int dx, dy)
```

ARGUMENTS

dest_region_return
 Returns the destination region.

dr_return
 Returns the result of the computation.

dx
dy Specify the x and y coordinates, which define the amount you want to move or shrink the specified region.

r Specifies the region.

rectangle
 Specifies the rectangle.

sra
srb Specify the two regions with which you want to perform the computation.

src_region
 Specifies the source region to be used.

DESCRIPTION

The *XIntersectRegion()* function computes the intersection of two regions.

The *XUnionRegion()* function computes the union of two regions.

The *XUnionRectWithRegion()* function updates the destination region from a union of the specified rectangle and the specified source region.

The *XSubtractRegion()* function subtracts *srb* from *sra* and stores the results in *dr_return*.

The *XXorRegion()* function calculates the difference between the union and intersection of two regions.

The *XOffsetRegion()* function moves the specified region by a specified amount.

The *XShrinkRegion()* function reduces the specified region by a specified amount. Positive values shrink the size of the region, and negative values expand the region.

SEE ALSO

XCreateRegion(), *XDrawRectangle()*, *XEmptyRegion()*

Xlib — C Language X Interface

NAME

XKeymapEvent – KeymapNotify event structure

STRUCTURES

The structure for KeymapNotify events contains:

generated on EnterWindow and FocusIn when KeymapState selected

```
typedef struct {
    int type;                KeymapNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;
    char key_vector[32];
} XKeymapEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *window* member is not used but is present to aid some toolkits. The *key_vector* member is set to the bit vector of the keyboard. Each bit set to 1 indicates that the corresponding key is currently pressed. The vector is represented as 32 bytes. Byte N (from 0) contains the bits for keys 8N to 8N + 7 with the least-significant bit in the byte representing key 8N.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XMapEvent(), *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XListFonts(), XFreeFontNames(), XListFontsWithInfo(), XFreeFontInfo() – obtain or free font names and information

SYNOPSIS

```
char **XListFonts (Display * display, char * pattern, int maxnames,
                  int * actual_count_return)
XFreeFontNames (char * list[])
char **XListFontsWithInfo (Display * display, char * pattern,
                          int maxnames, int * count_return,
                          XFontStruct ** info_return)
XFreeFontInfo (char ** names, XFontStruct * free_info, int actual_count)
```

ARGUMENTS

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

actual_count_return
Returns the actual number of font names.

count_return
Returns the actual number of matched font names.

display
Specifies the connection to the X server.

info_return
Returns the font information.

free_info
Specifies the font information returned by XListFontsWithInfo .

list
Specifies the array of strings you want to free.

maxnames
Specifies the maximum number of names to be returned.

names
Specifies the list of font names returned by XListFontsWithInfo .

pattern
Specifies the null-terminated pattern string that can contain wildcard characters.

DESCRIPTION

The *XListFonts()* function returns an array of available font names (as controlled by the font search path; see *XSetFontPath()*) that match the string you passed to the pattern argument. The pattern string can contain any characters, but each asterisk (*) is a wildcard for any number of characters, and each question mark (?) is a wildcard for a single character. If the pattern string is not in the Host Portable Character Encoding the result is implementation dependent. Use of uppercase or lowercase does not matter. Each returned string is null-terminated. If the data returned by the server is in the Latin Portable Character Encoding, then the returned strings are in the Host Portable Character Encoding. Otherwise, the result is implementation dependent. If there are no matching font names, *XListFonts()* returns NULL. The client should call *XFreeFontNames()* when finished with the result to free the memory.

The *XFreeFontNames()* function frees the array and strings returned by *XListFonts()* or *XListFontsWithInfo()*.

The *XListFontsWithInfo()* function returns a list of font names that match the specified pattern and their associated font information. The list of names is limited to size specified by *maxnames*. The information returned for each font is identical to what *XLoadQueryFont()* would return except that the per-character metrics are not returned.

The pattern string can contain any characters, but each asterisk (*) is a wildcard for any number of characters, and each question mark (?) is a wildcard for a single character. If the pattern string is not in the Host Portable Character Encoding the result is implementation dependent. Use of uppercase or lowercase does not matter. Each returned string is null-terminated. If the data returned by the server is in the Latin Portable Character Encoding, then the returned strings are in the Host Portable Character Encoding. Otherwise, the result is implementation dependent. If there are no matching font names, *XListFontsWithInfo()* returns NULL.

To free only the allocated name array, the client should call *XFreeFontNames()*. To free both the name array and the font information array, or to free just the font information array, the client should call *XFreeFontInfo()*.

The *XFreeFontInfo()* function frees the the font information array. To free an **XFontStruct** structure without closing the font, call *XFreeFontInfo()* with the *names* argument specified as NULL.

SEE ALSO

XLoadFont(), *XSetFontPath()*

Xlib — C Language X Interface

NAME

XLoadFont(), XQueryFont(), XLoadQueryFont(), XFreeFont(), XGetFontProperty(), XUnloadFont(), XCharStruct, XFontProp, XChar2b, XFontStruct – load or unload fonts and font metric structures

SYNOPSIS

```
Font XLoadFont (Display *display, char *name)
XFontStruct *XQueryFont (Display *display, XID font_ID)
XFontStruct *XLoadQueryFont (Display *display, char *name)
XFreeFont (Display *display, XFontStruct *font_struct)
Bool XGetFontProperty (XFontStruct *font_struct, Atom atom,
                      unsigned long *value_return)
XUnloadFont (Display *display, Font font)
```

ARGUMENTS

atom Specifies the atom for the property name you want returned.
display Specifies the connection to the X server.
font Specifies the font.
font_ID Specifies the font ID or the GContext ID.
font_struct Specifies the storage associated with the font.
gc Specifies the GC.
name Specifies the name of the font, which is a null-terminated string.
value_return Returns the value of the font property.

DESCRIPTION

The *XLoadFont()* function loads the specified font and returns its associated font ID. If the font name is not in the Host Portable Character Encoding the result is implementation dependent. Use of uppercase or lowercase does not matter. The interpretation of characters “?” and “*” in the name is not defined by the core protocol but is reserved for future definition. A structured format for font names is specified in the X Consortium standard *X Logical Font Description Conventions*. If *XLoadFont()* was unsuccessful at loading the specified font, a BadName error results. Fonts are not associated with a particular screen and can be stored as a component of any GC. When the font is no longer needed, call *XUnloadFont()*.

XLoadFont() can generate BadAlloc and BadName errors.

The *XQueryFont()* function returns a pointer to the **XFontStruct** structure, which contains information associated with the font. You can query a font or the font stored in a GC. The font ID stored in the **XFontStruct** structure will be the **GContext** ID, and you need to be careful when using this ID in other functions (see *XGContextFromGC()*). If the font does not exist, *XQueryFont()* returns NULL. To free this data, use *XFreeFontInfo()*.

XLoadQueryFont() can generate a BadAlloc error.

The *XLoadQueryFont()* function provides the most common way for accessing a font. *XLoadQueryFont()* both opens (loads) the specified font and returns a pointer to the appropriate **XFontStruct** structure. If the font name is not in the Host Portable Character Encoding the result is implementation dependent. If the font does not exist, *XLoadQueryFont()* returns NULL.

The *XFreeFont()* function deletes the association between the font resource ID and the specified font and frees the **XFontStruct** structure. The font itself will be freed when no other resource references it. The data and the font should not be referenced again.

XFreeFont() can generate a BadFont error.

Given the atom for that property, the *XGetFontProperty()* function returns the value of the specified font property. *XGetFontProperty()* also returns False if the property was not defined or True if it was defined. A set of predefined atoms exists for font properties, which can be found in **<X11/Xatom.h>**. This set contains the standard properties associated with a font. Although it is not guaranteed, it is likely that the predefined font properties will be present.

The *XUnloadFont()* function deletes the association between the font resource ID and the specified font. The font itself will be freed when no other resource references it. The font should not be referenced again.

XUnloadFont() can generate a BadFont error.

STRUCTURES

The **XFontStruct** structure contains all of the information for the font and consists of the font-specific information as well as a pointer to an array of **XCharStruct** structures for the characters contained in the font. The **XFontStruct**, **XFontProp**, and **XCharStruct** structures contain:

```
typedef struct {
    short lbearing;           origin to left edge of raster
    short rbearing;           origin to right edge of raster
    short width;              advance to next char's origin
    short ascent;             baseline to top edge of raster
    short descent;            baseline to bottom edge of raster
    unsigned short attributes; per char flags (not predefined)
} XCharStruct;

typedef struct {
    Atom name;
    unsigned long card32;
} XFontProp;

typedef struct {
    unsigned char byte1;      normal 16 bit characters are two bytes
    unsigned char byte2;
} XChar2b;

typedef struct {
    XExtData *ext_data;       hook for extension to hang data
    Font fid;                 Font id for this font
    unsigned direction;       hint about the direction font is painted
    unsigned min_char_or_byte2; first character
    unsigned max_char_or_byte2; last character
    unsigned min_byte1;       first row that exists
    unsigned max_byte1;       last row that exists
    Bool all_chars_exist;     flag if all characters have nonzero size
    unsigned default_char;    char to print for undefined character
    int n_properties;          how many properties there are
    XFontProp *properties;    pointer to array of additional properties
}
```

XCharStruct min_bounds;	minimum bounds over all existing char
XCharStruct max_bounds;	maximum bounds over all existing char
XCharStruct *per_char;	first_char to last_char information
int ascent;	logical extent above baseline for spacing
int descent;	logical decent below baseline for spacing

```
} XFontStruct;
```

X supports single byte/character, two bytes/character matrix, and 16-bit character text operations. Note that any of these forms can be used with a font, but a single byte/character text request can only specify a single byte (that is, the first row of a 2-byte font). You should view 2-byte fonts as a two-dimensional matrix of defined characters: byte1 specifies the range of defined rows and byte2 defines the range of defined columns of the font. Single byte/character fonts have one row defined, and the byte2 range specified in the structure defines a range of characters.

The bounding box of a character is defined by the **XCharStruct** of that character. When characters are absent from a font, the *default_char* is used. When fonts have all characters of the same size, only the information in the **XFontStruct** min and max bounds are used.

The members of the **XFontStruct** have the following semantics:

- The *direction* member can be either FontLeftToRight or FontRightToLeft. It is just a hint as to whether most **XCharStruct** elements have a positive (FontLeftToRight) or a negative (FontRightToLeft) character width metric. The core protocol defines no support for vertical text.
- If the *min_byte1* and *max_byte1* members are both zero, *min_char_or_byte2* specifies the linear character index corresponding to the first element of the *per_char* array, and *max_char_or_byte2* specifies the linear character index of the last element.

If either *min_byte1* or *max_byte1* are nonzero, both *min_char_or_byte2* and *max_char_or_byte2* are less than 256, and the 2-byte character index values corresponding to the *per_char* array element N (counting from 0) are:

```
byte1   = N/D + min_byte1
byte2   = N\D + min_char_or_byte2
```

where:

D = max_char_or_byte2 – min_char_or_byte2 + 1

/ = integer division

\ = integer modulus

- If the *per_char* pointer is NULL, all glyphs between the first and last character indexes inclusive have the same information, as given by both *min_bounds* and *max_bounds*.
- If *all_chars_exist* is True, all characters in the *per_char* array have nonzero bounding boxes.
- The *default_char* member specifies the character that will be used when an undefined or nonexistent character is printed. The *default_char* is a 16-bit character (not a 2-byte character). For a font using 2-byte matrix format, the *default_char* has byte1 in the most-significant byte and byte2 in the least-significant byte. If the *default_char* itself specifies an undefined or nonexistent character, no printing is performed for an

undefined or nonexistent character.

- The *min_bounds* and *max_bounds* members contain the most extreme values of each individual **XCharStruct** component over all elements of this array (and ignore nonexistent characters). The bounding box of the font (the smallest rectangle enclosing the shape obtained by superimposing all of the characters at the same origin [x,y]) has its upper-left coordinate at:

$$[x + \text{min_bounds.lbearing}, y - \text{max_bounds.ascent}]$$

Its width is:

$$\text{max_bounds.rbearing} - \text{min_bounds.lbearing}$$

Its height is:

$$\text{max_bounds.ascent} + \text{max_bounds.descent}$$

- The *ascent* member is the logical extent of the font above the baseline that is used for determining line spacing. Specific characters may extend beyond this.
- The *descent* member is the logical extent of the font at or below the baseline that is used for determining line spacing. Specific characters may extend beyond this.
- If the baseline is at Y-coordinate y, the logical extent of the font is inclusive between the Y-coordinate values (y – font.ascent) and (y + font.descent – 1). Typically, the minimum interline spacing between rows of text is given by *ascent* + *descent*.

For a character origin at [x,y], the bounding box of a character (that is, the smallest rectangle that encloses the character's shape) described in terms of **XCharStruct** components is a rectangle with its upper-left corner at:

$$[x + \text{lbearing}, y - \text{ascent}]$$

Its width is:

$$\text{rbearing} - \text{lbearing}$$

Its height is:

$$\text{ascent} + \text{descent}$$

The origin for the next character is defined to be:

$$[x + \text{width}, y]$$

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.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadFont

A value for a Font or GContext argument does not name a defined Font.

BadName

A font or color of the specified name does not exist.

SEE ALSO

XCreateGC(), *XListFonts()*, *XSetFontPath()*

Xlib — C Language X Interface

NAME

XLookupKeysym(), XRefreshKeyboardMapping(), XLookupString(), XRebindKeySym()
 – handle keyboard input events in Latin-1

SYNOPSIS

```
KeySym XLookupKeysym (XKeyEvent *key_event, int index)
XRefreshKeyboardMapping (XMappingEvent *event_map)
int XLookupString (XKeyEvent *event_struct, char *buffer_return,
                  int bytes_buffer, KeySym *keysym_return,
                  XComposeStatus *status_in_out)
XRebindKeysym (Display *display, KeySym keysym, KeySym list[],
               int mod_count, unsigned char *string, int num_bytes)
```

ARGUMENTS

buffer_return Returns the translated characters.

bytes_buffer Specifies the length of the buffer. No more than *bytes_buffer* of translation are returned.

num_bytes Specifies the number of bytes in the string argument.

display Specifies the connection to the X server.

event_map Specifies the mapping event that is to be used.

event_struct Specifies the key event structure to be used. You can pass **XKeyPressedEvent** or **XKeyReleasedEvent**.

index Specifies the index into the KeySyms list for the event's KeyCode.

key_event Specifies the KeyPress or KeyRelease event.

keysym Specifies the KeySym that is to be rebound.

keysym_return Returns the KeySym computed from the event if this argument is not NULL.

list Specifies the KeySyms to be used as modifiers.

mod_count Specifies the number of modifiers in the modifier list.

status_in_out Specifies or returns the **XComposeStatus** structure or NULL.

string Specifies the string that is copied and will be returned by *XLookupString()*.

DESCRIPTION

The *XLookupKeysym()* function uses a given keyboard event and the index you specified to return the KeySym from the list that corresponds to the KeyCode member in the **XKeyPressedEvent** or **XKeyReleasedEvent** structure. If no KeySym is defined for the KeyCode of the event, *XLookupKeysym()* returns NoSymbol.

The *XRefreshKeyboardMapping()* function refreshes the stored modifier and keymap information. You usually call this function when a MappingNotify event with a *request* member of MappingKeyboard or MappingModifier occurs. The result is to update Xlib's knowledge of the keyboard.

The *XLookupString()* function 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 (see *XRebindKeysym()*), the bound string will be stored in the buffer. Otherwise, the KeySym is mapped, if possible, to an ISO Latin-1 character or (if the Control modifier is on) to an ASCII control character, and that character is stored in the buffer. *XLookupString()* returns the number of characters that are stored in the buffer.

If present (non-NULL), the **XComposeStatus** structure records the state, which is private to Xlib, that needs preservation across calls to *XLookupString()* to implement compose processing. The creation of **XComposeStatus** structures is implementation dependent; a portable program must pass NULL for this argument.

The *XRebindKeysym()* function can be used to rebound the meaning of a KeySym for the client. It does not redefine any key in the X server but merely provides an easy way for long strings to be attached to keys. *XLookupString()* returns this string when the appropriate set of modifier keys are pressed and when the KeySym would have been used for the translation. No text conversions are performed; the client is responsible for supplying appropriately encoded strings. Note that you can rebound a KeySym that may not exist.

SEE ALSO

XButtonEvent(), *XMapEvent()*, *XStringToKeysym()*

Xlib — C Language X Interface

NAME

XMapEvent, XMappingEvent – MapNotify and MappingNotify event structures

STRUCTURES

The structure for MapNotify events contains:

```
typedef struct {
    int type;                MapNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window event;
    Window window;
    Bool override_redirect;  boolean, is override set...
} XMapEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *event* member is set either to the window that was mapped or to its parent, depending on whether **StructureNotify** or **SubstructureNotify** was selected. The *window* member is set to the window that was mapped. The *override_redirect* member is set to the override-redirect attribute of the window. Window manager clients normally should ignore this window if the override-redirect attribute is True, because these events usually are generated from pop-ups, which override structure control.

The structure for **MappingNotify** events is:

```
typedef struct {
    int type;                MappingNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;          unused
    int request;             one of MappingModifier, MappingKeyboard,
                             MappingPointer
    int first_keycode;       first keycode
    int count;               defines range of change w. first_keycode
} XMappingEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to **GraphicsExpose**. The *display* member is set to a pointer to the display the event was

read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *request* member is set to indicate the kind of mapping change that occurred and can be MappingModifier, MappingKeyboard, MappingPointer. If it is MappingModifier, the modifier mapping was changed. If it is MappingKeyboard, the keyboard mapping was changed. If it is MappingPointer, the pointer button mapping was changed. The *first_keycode* and *count* members are set only if the request member was set to MappingKeyboard. The number in *first_keycode* represents the first number in the range of the altered mapping, and *count* represents the number of keycodes altered.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XMapRequestEvent – MapRequest event structure

STRUCTURES

The structure for MapRequest events contains:

```
typedef struct {
    int type;                MapRequest
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window parent;
    Window window;
} XMapRequestEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *parent* member is set to the parent window. The *window* member is set to the window to be mapped.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XPropertyEvent()*, *XReparentEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XMapWindow(), XMapRaised(), XMapSubwindows() – map windows

SYNOPSIS

```
XMapWindow (Display *display, Window w)
XMapRaised (Display *display, Window w)
XMapSubwindows (Display *display, Window w)
```

ARGUMENTS

display Specifies the connection to the X server.
w Specifies the window.

DESCRIPTION

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

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

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

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

XMapWindow() can generate a BadWindow error.

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

XMapRaised() can generate a BadWindow error.

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

windows, rather than for each window.

XMapSubwindows() can generate a BadWindow error.

DIAGNOSTICS

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(), *XConfigureWindow()*, *XCreateWindow()*,
XDestroyWindow(), *XRaiseWindow()*, *XUnmapWindow()*

Xlib — C Language X Interface

NAME

XNextEvent(), XPeekEvent(), XWindowEvent(), XCheckWindowEvent(), XMaskEvent(), XCheckMaskEvent(), XCheckTypedEvent(), XCheckTypedWindowEvent() – select events by type

SYNOPSIS

```

XNextEvent (Display *display, XEvent *event_return)
XPeekEvent (Display *display, XEvent *event_return)
XWindowEvent (Display *display, Window w, long event_mask,
              XEvent *event_return)
Bool XCheckWindowEvent (Display *display, Window w,
                       long event_mask, XEvent *event_return)
XMaskEvent (Display *display, long event_mask,
           XEvent *event_return)
Bool XCheckMaskEvent (Display *display, long event_mask,
                    XEvent *event_return)
Bool XCheckTypedEvent (Display *display, int event_type,
                     XEvent *event_return)
Bool XCheckTypedWindowEvent (Display *display, Window w,
                           int event_type, XEvent *event_return)

```

ARGUMENTS

display Specifies the connection to the X server.

event_mask Specifies the event mask.

event_return Returns the matched event's associated structure.

event_return Returns the next event in the queue.

event_return Returns a copy of the matched event's associated structure.

event_type Specifies the event type to be compared.

w Specifies the window whose event you are interested in.

DESCRIPTION

The *XNextEvent()* function copies the first event from the event queue into the specified **XEvent** structure and then removes it from the queue. If the event queue is empty, *XNextEvent()* flushes the output buffer and blocks until an event is received.

The *XPeekEvent()* function returns the first event from the event queue, but it does not remove the event from the queue. If the queue is empty, *XPeekEvent()* flushes the output buffer and blocks until an event is received. It then copies the event into the client-supplied **XEvent** structure without removing it from the event queue.

The *XWindowEvent()* function searches the event queue for an event that matches both the specified window and event mask. When it finds a match, *XWindowEvent()* removes that event from the queue and copies it into the specified **XEvent** structure. The other events stored in the queue are not discarded. If a matching event is not in the queue, **XWindowEvent** flushes the output buffer and blocks until one is received.

The *XCheckWindowEvent()* function searches the event queue and then the events available on the server connection for the first event that matches the specified window and event mask. If it finds a match, *XCheckWindowEvent()* removes that event, copies it into the specified **XEvent** structure, and returns True. The other events stored in the queue are not discarded. If the event you requested is not available, *XCheckWindowEvent()* returns False, and the output buffer will have been flushed.

The *XMaskEvent()* function searches the event queue for the events associated with the specified mask. When it finds a match, *XMaskEvent()* removes that event and copies it into the specified **XEvent** structure. The other events stored in the queue are not discarded. If the event you requested is not in the queue, *XMaskEvent()* flushes the output buffer and blocks until one is received.

The *XCheckMaskEvent()* function searches the event queue and then any events available on the server connection for the first event that matches the specified mask. If it finds a match, *XCheckMaskEvent()* removes that event, copies it into the specified **XEvent** structure, and returns True. The other events stored in the queue are not discarded. If the event you requested is not available, *XCheckMaskEvent()* returns False, and the output buffer will have been flushed.

The *XCheckTypedEvent()* function searches the event queue and then any events available on the server connection for the first event that matches the specified type. If it finds a match, *XCheckTypedEvent()* removes that event, copies it into the specified **XEvent** structure, and returns True. The other events in the queue are not discarded. If the event is not available, *XCheckTypedEvent()* returns False, and the output buffer will have been flushed.

The *XCheckTypedWindowEvent()* function searches the event queue and then any events available on the server connection for the first event that matches the specified type and window. If it finds a match, *XCheckTypedWindowEvent()* removes the event from the queue, copies it into the specified **XEvent** structure, and returns True. The other events in the queue are not discarded. If the event is not available, *XCheckTypedWindowEvent()* returns False, and the output buffer will have been flushed.

SEE ALSO

XAnyEvent(), *XIfEvent()*, *XPutBackEvent()*, *XSendEvent()*

Xlib — C Language X Interface

NAME

XNoOp() – No Operation

SYNOPSIS

XNoOp (Display **display*)

ARGUMENTS

display Specifies the connection to the X server.

DESCRIPTION

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

SEE ALSO

Xlib — C Language X Interface

NAME

XOpenDisplay(), XCloseDisplay() – connect or disconnect to X server

SYNOPSIS

```
Display *XOpenDisplay (char *display_name)
XCloseDisplay (Display *display)
```

ARGUMENTS

display Specifies the connection to the X server.

display_name

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

DESCRIPTION

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

If successful, *XOpenDisplay()* returns a pointer to a **Display** structure, which is defined in *<X11/Xlib.h>*. If *XOpenDisplay()* does not succeed, it returns NULL. After a successful call to *XOpenDisplay()*, all of the screens in the display can be used by the client. The screen number specified in the *display_name* argument is returned by the *DefaultScreen()* macro (or the *XDefaultScreen()* function). You can access elements of the **Display** and **Screen** structures only by using the information macros or functions. For information about using macros and functions to obtain information from the **Display** structure, see section 2.2.1.

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

XCloseDisplay() can generate a BadGC error.

SEE ALSO

AllPlanes(), *XFlush()*, *XSetCloseDownMode()*

Xlib — C Language X Interface

NAME

XOpenIM(), XCloseIM(), XGetIMValues(), XDisplayOfIM(), XLocaleOfIM() – open, close, and obtain input method information

SYNOPSIS

```
XIM XOpenIM (Display *display, XrmDataBase db, char *res_name,
             char *res_class)
Status XCloseIM (XIM im)
char * XGetIMValues (XIM im, ...)
Display * XDisplayOfIM (XIM im)
char * XLocaleOfIM (XIM im)
```

ARGUMENTS

db Specifies a pointer to the resource database.
display Specifies the connection to the X server.
im Specifies the input method.
res_class Specifies the full class name of the application.
res_name Specifies the full resource name of the application.
... Specifies the variable length argument list to get XIM values.

DESCRIPTION

The *XOpenIM()* function opens an input method, matching the current locale and modifiers specification. Current locale and modifiers are bound to the input method at opening time. The locale associated with an input method cannot be changed dynamically. This implies the strings returned by *XmbLookupString()* or *XwcLookupString()*, for any input context affiliated with a given input method, will be encoded in the locale current at the time input method is opened.

The specific input method to which this call will be routed is identified on the basis of the current locale. *XOpenIM()* will identify a default input method corresponding to the current locale. That default can be modified using *XSetLocaleModifiers()* for the input method modifier.

The *db* argument is the resource database to be used by the input method for looking up resources that are private to the input method. It is not intended that this database be used to look up values that can be set as IC values in an input context. If *db* is NULL, no data base is passed to the input method.

The *res_name* and *res_class* arguments specify the resource name and class of the application. They are intended to be used as prefixes by the input method when looking up resources that are common to all input contexts that may be created for this input method. The characters used for resource names and classes must be in the X portable character set. The resources looked up are not fully specified if *res_name* or *res_class* is NULL.

The *res_name* and *res_class* arguments are not assumed to exist beyond the call to *XOpenIM()*. The specified resource database is assumed to exist for the lifetime of the input method.

XOpenIM() returns NULL if no input method could be opened.

The *XCLOSEIM()* function closes the specified input method.

The *XGETIMVALUES()* function presents a variable argument list programming interface for querying properties or features of the specified input method. This function returns NULL if it succeeds; otherwise, it returns the name of the first argument that could not be obtained.

Only one standard argument is defined by Xlib: *XNQUERYINPUTSTYLE*, which must be used to query about input styles supported by the input method.

A client should always query the input method to determine which styles are supported. The client should then find an input style it is capable of supporting.

If the client cannot find an input style that it can support it should negotiate with the user the continuation of the program (exit, choose another input method, and so on).

The argument value must be a pointer to a location where the returned value will be stored. The returned value is a pointer to a structure of type **XIMStyles**. Clients are responsible for freeing the **XIMStyles** data structure. To do so, use *XFree()*.

The *XDISPLAYOFIM()* function returns the display associated with the specified input method.

The *XLCALEOFIM()* returns the locale associated with the specified input method.

SEE ALSO

XCreateIC(), *XSetICFocus()*, *XSetICValues()*, *XmbResetIC()*

Xlib — C Language X Interface

NAME

XParseGeometry(), XWMGeometry() – parse window geometry

SYNOPSIS

```
int XParseGeometry (char *parsestring, int *x_return,
                   int *y_return, unsigned int *width_return,
                   unsigned int *height_return)
int XWMGeometry (Display *display, int screen, char *user_geom,
                char *def_geom, unsigned int bwidth,
                XSizeHints *hints, int *x_return, int *y_return,
                int *width_return, int *height_return,
                int *gravity_return)
```

ARGUMENTS

position
default_position Specify the geometry specifications.
display Specifies the connection to the X server.
fheight
fwidth Specify the font height and width in pixels (increment size).
parsestring Specifies the string you want to parse.
screen Specifies the screen.
width_return
height_return Return the width and height determined.
xadder
yadder Specify additional interior padding needed in the window.
x_return
y_return Return the x and y offsets.
bwidth Specifies the border width.
hints Specifies the size hints for the window in its normal state.
def_geom Specifies the application's default geometry or NULL.
gravity_return Returns the window gravity.
user_geom Specifies the user-specified geometry or NULL.

DESCRIPTION

By convention, X applications use a standard string to indicate window size and placement. XParseGeometry makes it easier to conform to this standard because it allows you to parse the standard window geometry. Specifically, this function lets you parse strings of the form:

```
[ = ] [ < width > { xX } < height > ] [ { + - } < xoffset > { + - } < yoffset > ]
```

The fields map into the arguments associated with this function. (Items enclosed in < > are integers, items in [] are optional, and items enclosed in { } indicate “choose one of.” Note that the brackets should not appear in the actual string.) If the string is not in the Host Portable Character Encoding the result is implementation dependent.

The *XParseGeometry()* function returns a bitmask that indicates which of the four values (width, height, xoffset, and yoffset) were actually found in the string and whether the x and y values are negative. By convention, -0 is not equal to +0, because the user needs to be able to say “position the window relative to the right or bottom edge.” For each value found, the corresponding argument is updated. For each value not found, the argument is left unchanged. The bits are represented by XValue, YValue, WidthValue, HeightValue, XNegative, or YNegative and are defined in **<X11/Xutil.h>**. They will be set whenever one of the values is defined or one of the signs is set.

If the function returns either the XValue or YValue flag, you should place the window at the requested position.

The *XWMGeometry()* function combines any geometry information (given in the format used by *XParseGeometry()*) specified by the user and by the calling program with size hints (usually the ones to be stored in WM_NORMAL_HINTS) and returns the position, size, and gravity (NorthWestGravity, NorthEastGravity, SouthEastGravity, or SouthWestGravity) that describe the window. If the base size is not set in the **XSizeHints** structure, the minimum size is used if set. Otherwise, a base size of zero is assumed. If no minimum size is set in the hints structure, the base size is used. A mask (in the form returned by *XParseGeometry()*) that describes which values came from the user specification and whether or not the position coordinates are relative to the right and bottom edges is returned. Note that these coordinates will have already been accounted for in the *x_return* and *y_return* values.

Note that invalid geometry specifications can cause a width or height of zero to be returned. The caller may pass the address of the hints *win_gravity* field as *gravity_return* to update the hints directly.

SEE ALSO

XSetWMProperties()

Xlib — C Language X Interface

NAME

XPolygonRegion(), XClipBox() – generate regions

SYNOPSIS

```
Region XPolygonRegion (XPoint points[], int n, int fill_rule)  
XClipBox (Region r, XRectangle *rect_return)
```

ARGUMENTS

fill_rule Specifies the fill-rule you want to set for the specified GC. You can pass EvenOddRule or WindingRule.
n Specifies the number of points in the polygon.
points Specifies an array of points.
r Specifies the region.
rect_return Returns the smallest enclosing rectangle.

DESCRIPTION

The *XPolygonRegion()* function returns a region for the polygon defined by the points array. For an explanation of *fill_rule*, see *XCreateGC()*.

The *XClipBox()* function returns the smallest rectangle enclosing the specified region.

SEE ALSO

XCreateGC(), *XDrawPoint()*, *XDrawRectangle()*

Xlib — C Language X Interface

NAME

XPropertyEvent – PropertyNotify event structure

STRUCTURES

The structure for PropertyNotify events contains:

```
typedef struct {
    int type;                PropertyNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;
    Atom atom;
    Time time;
    int state;               PropertyNewValue or PropertyDelete
} XPropertyEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the type member set to **GraphicsExpose**. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *window* member is set to the window whose associated property was changed. The *atom* member is set to the property's atom and indicates which property was changed or desired. The *time* member is set to the server time when the property was changed. The *state* member is set to indicate whether the property was changed to a new value or deleted and can be PropertyNewValue or PropertyDelete. The *state* member is set to PropertyNewValue when a property of the window is changed using *XChangeProperty()* or *XRotateWindowProperties()* (even when adding zero-length data using *XChangeProperty()*) and when replacing all or part of a property with identical data using *XChangeProperty()* or *XRotateWindowProperties()*. The *state* member is set to PropertyDelete when a property of the window is deleted using *XDeleteProperty* or, if the delete argument is True, *XGetWindowProperty*.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGetWindowProperty()*,
XGraphicsExposeEvent(), *XGravityEvent()*, *XKeymapEvent()*, *XMapEvent()*,
XMapRequestEvent(), *XReparentEvent()*, *XResizeRequestEvent()*, *XSelectionClearEvent()*,
XSelectionEvent(), *XSelectionRequestEvent()*, *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XPutBackEvent() – put events back on the queue

SYNOPSIS

XPutBackEvent (Display **display*, XEvent **event*)

ARGUMENTS

display Specifies the connection to the X server.
event Specifies the event.

DESCRIPTION

The *XPutBackEvent()* function pushes an event back onto the head of the display's event queue by copying the event into the queue. This can be useful if you read an event and then decide that you would rather deal with it later. There is no limit to the number of times in succession that you can call *XPutBackEvent()*.

SEE ALSO

XAnyEvent(), *XIfEvent()*, *XNextEvent()*, *XSendEvent()*

Xlib — C Language X Interface

NAME

XPutImage(), XGetImage(), XGetSubImage() – transfer images

SYNOPSIS

```
XPutImage (Display *display, Drawable d, GC gc,
           XImage *image, int src_x, src_y, int dest_x,
           int dest_y, unsigned int width, height)
XImage *XGetImage (Display *display, Drawable d, int x,
                  int y, unsigned int width, height,
                  unsigned long plane_mask, int format)
XImage *XGetSubImage (Display *display, Drawable d, int x,
                     int y, unsigned int width, int height,
                     unsigned long plane_mask, int format,
                     XImage *dest_image, int dest_x,
                     int dest_y)
```

ARGUMENTS

d Specifies the drawable.

dest_image Specify the destination image.

dest_x

dest_y Specify the x and y coordinates, which are relative to the origin of the drawable and are the coordinates of the subimage or which are relative to the origin of the destination rectangle, specify its upper-left corner, and determine where the subimage is placed in the destination image.

display Specifies the connection to the X server.

format Specifies the format for the image. You can pass XYPixmap or ZPixmap.

gc Specifies the GC.

image Specifies the image you want combined with the rectangle.

plane_mask Specifies the plane mask.

src_x Specifies the offset in X from the left edge of the image defined by the XImage data structure.

src_y Specifies the offset in Y from the top edge of the image defined by the XImage data structure.

width

height Specify the width and height of the subimage, which define the dimensions of the rectangle.

x

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

DESCRIPTION

The *XPutImage()* function combines an image in memory with a rectangle of the specified drawable. If XYBitmap format is used, the depth of the image must be one, or a BadMatch error results. The foreground pixel in the GC defines the source for the one bits in the image, and the background pixel defines the source for the zero bits. For XYPixmap and ZPixmap, the depth of the image must match the depth of the drawable, or a BadMatch error results. The section of the image defined by the *src_x*, *src_y*, *width*, and *height* arguments is drawn on the specified part of the drawable.

This function uses these GC components: function, plane-mask, subwindow-mode, clip-x-origin, clip-y-origin, and clip-mask. It also uses these GC mode-dependent components: foreground and background.

XPutImage() can generate BadDrawable, BadGC, BadMatch, and BadValue errors.

The *XGetImage()* function returns a pointer to an **XImage** structure. This structure provides you with the contents of the specified rectangle of the drawable in the format you specify. If the format argument is XYPixmap, the image contains only the bit planes you passed to the *plane_mask* argument. If the *plane_mask* argument only requests a subset of the planes of the display, the depth of the returned image will be the number of planes requested. If the format argument is ZPixmap, *XGetImage()* returns as zero the bits in all planes not specified in the *plane_mask* argument. The function performs no range checking on the values in *plane_mask* and ignores extraneous bits.

XGetImage() returns the depth of the image to the depth member of the **XImage** structure. The depth of the image is as specified when the drawable was created, except when getting a subset of the planes in XYPixmap format, when the depth is given by the number of bits set to 1 in *plane_mask*.

If the drawable is a pixmap, the given rectangle must be wholly contained within the pixmap, or a BadMatch error results. If the drawable is a window, the window must be viewable, and it must be the case that if there were no inferiors or overlapping windows, the specified rectangle of the window would be fully visible on the screen and wholly contained within the outside edges of the window, or a BadMatch error results. Note that the borders of the window can be included and read with this request. If the window has backing-store, the backing-store contents are returned for regions of the window that are obscured by noninferior windows. If the window does not have backing-store, the returned contents of such obscured regions are undefined. The returned contents of visible regions of inferiors of a different depth than the specified window's depth are also undefined. The pointer cursor image is not included in the returned contents. If a problem occurs, *XGetImage()* returns NULL.

XGetImage() can generate BadDrawable, BadMatch, and BadValue errors.

The *XGetSubImage()* function updates *dest_image* with the specified subimage in the same manner as *XGetImage()*. If the format argument is XYPixmap, the image contains only the bit planes you passed to the *plane_mask* argument. If the format argument is ZPixmap, *XGetSubImage()* returns as zero the bits in all planes not specified in the *plane_mask* argument. The function performs no range checking on the values in *plane_mask* and ignores extraneous bits. As a convenience, *XGetSubImage()* returns a pointer to the same **XImage** structure specified by *dest_image*.

The depth of the destination **XImage** structure must be the same as that of the drawable. If the specified subimage does not fit at the specified location on the destination image, the right and bottom edges are clipped. If the drawable is a pixmap, the given rectangle must be wholly contained within the pixmap, or a BadMatch error results. If the drawable is a window, the window must be viewable, and it must be the case that if there were no inferiors or overlapping windows, the specified rectangle of the window would be fully visible on the screen and wholly contained within the outside edges of the window, or a BadMatch error results. If the window has backing-store, then the backing-store contents are returned for regions of the window that are obscured by noninferior windows. If the window does not have backing-store, the returned contents of such obscured regions are undefined. The returned contents of visible regions of inferiors of a different depth than the specified window's depth are also undefined. If a

problem occurs, *XGetSubImage()* returns NULL.

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

DIAGNOSTICS

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

An InputOnly window is used as a Drawable.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

Xlib — C Language X Interface

NAME

XQueryBestSize(), XQueryBestTile(), XQueryBestStipple() – determine efficient sizes

SYNOPSIS

```
Status XQueryBestSize (Display *display, int class,
                      Drawable which_screen, unsigned int width,
                      unsigned int height,
                      unsigned int *width_return,
                      unsigned int *height_return)
Status XQueryBestTile (Display *display, Drawable which_screen,
                      unsigned int width, height,
                      unsigned int *width_return,
                      unsigned int *height_return)
Status XQueryBestStipple (Display *display, Drawable which_screen,
                          unsigned int width, unsigned int height,
                          unsigned int *width_return,
                          unsigned int *height_return)
```

ARGUMENTS

class Specifies the class that you are interested in. You can pass TileShape, CursorShape, or StippleShape.

display Specifies the connection to the X server.

width

height Specify the width and height.

which_screen Specifies any drawable on the screen.

width_return

height_return Return the width and height of the object best supported by the display hardware.

DESCRIPTION

The *XQueryBestSize()* function returns the best or closest size to the specified size. For CursorShape, this is the largest size that can be fully displayed on the screen specified by *which_screen*. For TileShape, this is the size that can be tiled fastest. For StippleShape, this is the size that can be stippled fastest. For CursorShape, the drawable indicates the desired screen. For TileShape and StippleShape, the drawable indicates the screen and possibly the window class and depth. An InputOnly window cannot be used as the drawable for TileShape or StippleShape, or a BadMatch error results.

XQueryBestSize() can generate BadDrawable, BadMatch, and BadValue errors.

The *XQueryBestTile()* function returns the best or closest size, that is, the size that can be tiled fastest on the screen specified by *which_screen*. The drawable indicates the screen and possibly the window class and depth. If an InputOnly window is used as the drawable, a BadMatch error results.

XQueryBestTile() can generate BadDrawable and BadMatch errors.

The *XQueryBestStipple()* function returns the best or closest size, that is, the size that can be stippled fastest on the screen specified by *which_screen*. The drawable indicates the screen and possibly the window class and depth. If an InputOnly window is used as the drawable, a BadMatch error results.

XQueryBestStipple() can generate BadDrawable and BadMatch errors.

DIAGNOSTICS

BadMatch

An InputOnly window is used as a Drawable.

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadMatch

The values do not exist for an InputOnly window.

BadValue

Some numeric value falls outside the range of values accepted by the request.

Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XCreateGC(), *XSetArcMode()*, *XSetClipOrigin()*, *XSetFillStyle()*, *XSetFont()*,
XSetLineAttributes(), *XSetState()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XQueryColor(), XQueryColors(), XLookupColor(), XParseColor() – obtain color values

SYNOPSIS

```
XQueryColor (Display *display, Colormap colormap,
             XColor *def_in_out)
XQueryColors (Display *display, Colormap colormap,
             XColor defs_in_out[], int ncolors)
Status XLookupColor (Display *display, Colormap colormap,
                   char *color_name, XColor *exact_def_return,
                   XColor *screen_def_return)
Status XParseColor (Display *display, Colormap colormap,
                  char *spec, XColor *exact_def_return)
```

ARGUMENTS

colormap
Specifies the colormap.

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

def_in_out
Specifies and returns the RGB values for the pixel specified in the structure.

defs_in_out
Specifies and returns an array of color definition structures for the pixel specified in the structure.

display
Specifies the connection to the X server.

exact_def_return
Returns the exact RGB values.

ncolors
Specifies the number of **XColor** structures in the color definition array.

screen_def_return
Returns the closest RGB values provided by the hardware.

spec
Specifies the color name string; case is ignored.

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

DESCRIPTION

The *XQueryColor()* function returns the current RGB value for the pixel in the **XColor** structure and sets the *DoRed*, *DoGreen*, and *DoBlue* flags. The *XQueryColors()* function returns the RGB value for each pixel in each **XColor** structure, and sets the *DoRed*, *DoGreen*, and *DoBlue* flags in each structure.

XQueryColor() and *XQueryColors()* can generate BadColor and BadValue errors.

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

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

XLookupColor() and *XParseColor()* can generate BadColor error.

COLOR NAMES

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

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

```
<red>, <green>, <blue> := h | hh | hhh | hhhh
```

```
h := single hexadecimal digits (case insignificant)
```

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

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

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

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

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

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

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

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

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

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

DIAGNOSTICS

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XAllocColor(), *XCreateColormap()*, *XStoreColors()*

Xlib — C Language X Interface

NAME

XQueryPointer() – get pointer coordinates

SYNOPSIS

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

ARGUMENTS

child_return

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

display Specifies the connection to the X server.

mask_return

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

root_return

Returns the root window that the pointer is in.

root_x_return

root_y_return

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

w

Specifies the window.

win_x_return

win_y_return

Return the pointer coordinates relative to the specified window.

DESCRIPTION

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

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

XQueryPointer() can generate a BadWindow error.

DIAGNOSTICS

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XGetWindowAttributes(), *XQueryTree()*

Xlib — C Language X Interface

NAME

XQueryTree() – query window tree information

SYNOPSIS

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

ARGUMENTS

children_return
Returns the list of children.

display Specifies the connection to the X server.

nchildren_return
Returns the number of children.

parent_return
Returns the parent window.

root_return
Returns the root window.

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

DESCRIPTION

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

BUGS

This really should return a **screen ***, not a root window ID.

SEE ALSO

XFree(), *XGetWindowAttributes()*, *XQueryPointer()*
Xlib — C Language X Interface

NAME

XRaiseWindow(), XLowerWindow(), XCirculateSubwindows(),
XCirculateSubwindowsUp(), XCirculateSubwindowsDown(), XRestackWindows() –
change window stacking order

SYNOPSIS

```
XRaiseWindow (Display *display, Window w)
XLowerWindow (Display *display, Window w)
XCirculateSubwindows (Display *display, Window w,
                     int direction)
XCirculateSubwindowsUp (Display *display, Window w)
XCirculateSubwindowsDown (Display *display, Window w)
XRestackWindows (Display *display, Window windows[],
                 int nwindows)
```

ARGUMENTS

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

display Specifies the connection to the X server.

nwindows Specifies the number of windows to be restacked.

w Specifies the window.

windows Specifies an array containing the windows to be restacked.

DESCRIPTION

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

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

XRaiseWindow() can generate a BadWindow error.

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

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

XLowerWindow() can generate a *BadWindow* error.

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

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

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

XCirculateSubwindowsUp() can generate a *BadWindow* error.

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

XCirculateSubwindowsDown() can generate a *BadWindow* error.

The *XRestackWindows()* function restacks the windows in the order specified, from top to bottom. The stacking order of the first window in the windows array is unaffected, but the other windows in the array are stacked underneath the first window, in the order of the array. The stacking order of the other windows is not affected. For each window in the window array that is not a child of the specified window, a *BadMatch* error results.

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

XRestackWindows() can generate *BadWindow* error.

DIAGNOSTICS

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a *Window* argument does not name a defined *Window*.

SEE ALSO

XChangeWindowAttributes(), *XConfigureWindow()*, *XCreateWindow()*, *XDestroyWindow()*, *XMapWindow()*, *XUnmapWindow()*

NAME

XReadBitmapFile(), XWriteBitmapFile(), XCreatePixmapFromBitmapData(),
XCreateBitmapFromData() – manipulate bitmaps

SYNOPSIS

```
int XReadBitmapFile (Display *display, Drawable d,
                    char *filename, unsigned int *width_return,
                    unsigned int *height_return,
                    Pixmap *bitmap_return, int *x_hot_return,
                    int *y_hot_return)
int XWriteBitmapFile (Display *display, char *filename,
                    Pixmap bitmap, unsigned int width,
                    unsigned int height, int x_hot,
                    int y_hot)
Pixmap XCreatePixmapFromBitmapData (Display *display, Drawable d,
                                    char *data,
                                    unsigned int width,
                                    unsigned int height,
                                    unsigned long fg,
                                    unsigned long bg,
                                    unsigned int depth)
Pixmap XCreateBitmapFromData (Display *display, Drawable d,
                              char *data, unsigned int width,
                              unsigned int height)
```

ARGUMENTS

bitmap Specifies the bitmap.
bitmap_return Returns the bitmap that is created.
d Specifies the drawable that indicates the screen.
data Specifies the data in bitmap format.
data Specifies the location of the bitmap data.
depth Specifies the depth of the pixmap.
display Specifies the connection to the X server.
fg
bg Specify the foreground and background pixel values to use.
filename Specifies the file name to use. The format of the file name is operating-system dependent.
width
height Specify the width and height.
width_return
height_return Return the width and height values of the read in bitmap file.
x_hot
y_hot Specify where to place the hotspot coordinates (or -1,-1 if none are present) in the file.
x_hot_return
y_hot_return Return the hotspot coordinates.

DESCRIPTION

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

XReadBitmapFile() returns the bitmap's height and width, as read from the file, to *width_return* and *height_return*. It then creates a pixmap of the appropriate size, reads the bitmap data from the file into the pixmap, and assigns the pixmap to the caller's variable *bitmap*. The caller must free the bitmap using *XFreePixmap()* when finished. If *<name>_x_hot* and *<name>_y_hot* exist, *XReadBitmapFile()* returns them to *x_hot_return* and *y_hot_return*; otherwise, it returns *-1,-1*.

XReadBitmapFile() can generate *BadAlloc* and *BadDrawable* errors.

The *XWriteBitmapFile()* function writes a bitmap out to a file in the X version 11 format. The file is written in the encoding of the current locale. If the file cannot be opened for writing, it returns *BitmapOpenFailed*. If insufficient memory is allocated, *XWriteBitmapFile()* returns *BitmapNoMemory*; otherwise, on no error, it returns *BitmapSuccess*. If *x_hot* and *y_hot* are not *-1,-1*, *XWriteBitmapFile()* writes them out as the hotspot coordinates for the bitmap.

XWriteBitmapFile() can generate *BadDrawable* and *BadMatch* errors.

The *XCreatePixmapFromBitmapData()* function creates a pixmap of the given depth and then does a bitmap-format *XPutImage()* of the data into it. The depth must be supported by the screen of the specified drawable, or a *BadMatch* error results.

XCreatePixmapFromBitmapData() can generate *BadAlloc* and *BadMatch* errors.

The *XCreateBitmapFromData()* function allows you to include in your C program (using *#include*) a bitmap file that was written out by *XWriteBitmapFile()* (X version 11 format only) without reading in the bitmap file. The following example creates a gray bitmap:

```
#include "gray.bitmap"

Pixmap bitmap;
bitmap = XCreateBitmapFromData(display, window,
                               gray_bits, gray_width, gray_height);
```

If insufficient working storage was allocated, *XCreateBitmapFromData()* returns *None*. It is your responsibility to free the bitmap using *XFreePixmap()* when finished.

XCreateBitmapFromData() can generate a *BadAlloc* error.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

BadMatch

An *InputOnly* window is used as a Drawable.

SEE ALSO

XCreatePixmap(), *XPutImage()*

NAME

XRecolorCursor(), XFreeCursor(), XQueryBestCursor() – manipulate cursors

SYNOPSIS

```
XRecolorCursor (Display *display, Cursor cursor,
                XColor *foreground_color,
                XColor *background_color)
XFreeCursor (Display *display, Cursor cursor)
Status XQueryBestCursor (Display *display, Drawable d,
                        unsigned int width,
                        unsigned int height,
                        unsigned int *width_return,
                        unsigned int *height_return)
```

ARGUMENTS

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

cursor
Specifies the cursor.

d
Specifies the drawable, which indicates the screen.

display
Specifies the connection to the X server.

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

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

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

DESCRIPTION

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

XRecolorCursor() can generate a BadCursor error.

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

XFreeCursor() can generate a BadCursor error.

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

XQueryBestCursor() can generate a BadDrawable error.

DIAGNOSTICS

BadCursor
A value for a Cursor argument does not name a defined Cursor.

BadDrawable

A value for a Drawable argument does not name a defined Window or Pixmap.

SEE ALSO

XCreateColormap(), *XCreateFontCursor()*, *XDefineCusor()*

Xlib — C Language X Interface

NAME

XReparentEvent – ReparentNotify event structure

STRUCTURES

The structure for ReparentNotify events contains:

```
typedef struct {
    int type;                ReparentNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window event;
    Window window;
    Window parent;
    int x, y;
    Bool override_redirect;
} XReparentEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to **GraphicsExpose**. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *event* member is set either to the reparented window or to the old or the new parent, depending on whether StructureNotify or SubstructureNotify was selected. The *window* member is set to the window that was reparented. The *parent* member is set to the new parent window. The *x* and *y* members are set to the reparented window's coordinates relative to the new parent window's origin and define the upper-left outer corner of the reparented window. The *override_redirect* member is set to the override-redirect attribute of the window specified by the window member. Window manager clients normally should ignore this window if the *override_redirect* member is True.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XResizeRequestEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XReparentWindow() – reparent windows

SYNOPSIS

```
XReparentWindow (Display *display, Window w, Window parent,
                 int x, int y)
```

ARGUMENTS

display Specifies the connection to the X server.
parent Specifies the parent window.
w Specifies the window.
x
y Specify the x and y 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 *override_redirect* member returned in this event is set to the window's corresponding attribute. Window manager clients usually should ignore this window if this member is set to True. Finally, if the specified window was originally mapped, the X server automatically performs a MapWindow request on it.

The X server performs normal exposure processing on formerly obscured windows. The X server might not generate Expose events for regions from the initial UnmapWindow request that are immediately obscured by the final MapWindow request. A BadMatch error results if:

- The new parent window is not on the same screen as the old parent window.
- The new parent window is the specified window or an inferior of the specified window.
- The new parent is InputOnly and the window is not.
- The specified window has a ParentRelative background, and the new parent window is not the same depth as the specified window.

XReparentWindow() can generate BadMatch and BadWindow errors.

DIAGNOSTICS

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeSaveSet()

Xlib — C Language X Interface

NAME

XResizeRequestEvent – ResizeRequest event structure

STRUCTURES

The structure for ResizeRequest events contains:

```
typedef struct {
    int type;                ResizeRequest
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;
    int width, height;
} XResizeRequestEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *window* member is set to the window whose size another client attempted to change. The *width* and *height* members are set to the inside size of the window, excluding the border.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XReparentEvent(), *XSelectionClearEvent()*, *XSelectionEvent()*, *XSelectionRequestEvent()*,
XUnmapEvent(), *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XResourceManagerString(), XScreenResourceString() – obtain server resource properties

SYNOPSIS

```
char *XResourceManagerString (Display *display)
char *XScreenResourceString (Screen *screen)
```

ARGUMENTS

display Specifies the connection to the X server.
screen Specifies the screen.

DESCRIPTION

The *XResourceManagerString()* returns the RESOURCE_MANAGER property from the server's root window of screen zero, which was returned when the connection was opened using *XOpenDisplay()*. The property is converted from type STRING to the current locale. The conversion is identical to that produced by *XmbTextPropertyToTextList()* for a singleton STRING property. The returned string is owned by Xlib, and should not be freed by the client. Note that the property value must be in a format that is acceptable to *XrmGetStringDatabase()*. If no property exists, NULL is returned.

The *XStringResourceString()* returns the SCREEN_RESOURCES property from the root window of the specified screen. The property is converted from type STRING to the current locale. The conversion is identical to that produced by *XmbTextPropertyToTextList()* for a singleton STRING property. Note that the property value must be in a format that is acceptable to *XrmGetStringDatabase()*. If no property exists, NULL is returned. The caller is responsible for freeing the returned string, using *XFree()*.

SEE ALSO

Xlib — C Language X Interface

NAME

XSaveContext(), XFindContext(), XDeleteContext(), XUniqueContext() – associative look-up routines

SYNOPSIS

```
int XSaveContext (Display *display, XID rid, XContext context,
                 XPointer data)
int XFindContext (Display *display, XID rid, XContext context,
                 XPointer *data_return)
int XDeleteContext (Display *display, XID rid, XContext context)
XContext XUniqueContext (void)
```

ARGUMENTS

context Specifies the context type to which the data belongs.
data Specifies the data to be associated with the window and type.
data_return Returns the data.
display Specifies the connection to the X server.
rid Specifies the resource ID with which the data is associated.

DESCRIPTION

If an entry with the specified resource ID and type already exists, *XSaveContext()* overrides it with the specified context. The *XSaveContext()* function returns a nonzero error code if an error has occurred and zero otherwise. Possible errors are XCENOMEM (out of memory).

Because it is a return value, the data is a pointer. The *XFindContext()* function returns a nonzero error code if an error has occurred and zero otherwise. Possible errors are XCNOENT (context-not-found).

The *XDeleteContext()* function deletes the entry for the given resource ID and type from the data structure. This function returns the same error codes that *XFindContext()* returns if called with the same arguments. *XDeleteContext()* does not free the data whose address was saved.

The *XUniqueContext()* function creates a unique context type that may be used in subsequent calls to *XSaveContext()*.

SEE ALSO

Xlib — C Language X Interface

NAME

XSelectInput() – select input events

SYNOPSIS

XSelectInput (Display **display*, Window *w*, long *event_mask*)

ARGUMENTS

display Specifies the connection to the X server.
event_mask Specifies the event mask.
w Specifies the window whose events you are interested in.

DESCRIPTION

The *XSelectInput()* function requests that the X server report the events associated with the specified event mask. Initially, X will not report any of these events. Events are reported relative to a window. If a window is not interested in a device event, it usually propagates to the closest ancestor that is interested, unless the *do_not_propagate_mask* prohibits it.

Setting the event-mask attribute of a window overrides any previous call for the same window but not for other clients. Multiple clients can select for the same events on the same window with the following restrictions:

- Multiple clients can select events on the same window because their event masks are disjoint. When the X server generates an event, it reports it to all interested clients.
- Only one client at a time can select CirculateRequest, ConfigureRequest, or MapRequest events, which are associated with the event mask SubstructureRedirectMask.
- Only one client at a time can select a ResizeRequest event, which is associated with the event mask ResizeRedirectMask.
- Only one client at a time can select a ButtonPress event, which is associated with the event mask ButtonPressMask.

The server reports the event to all interested clients.

XSelectInput() can generate a BadWindow error.

DIAGNOSTICS

BadWindow
A value for a Window argument does not name a defined Window.

SEE ALSO

Xlib — C Language X Interface

NAME

XSelectionClearEvent – SelectionClear event structure

STRUCTURES

The structure for SelectionClear events contains:

```
typedef struct {
    int type;                SelectionClear
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;
    Atom selection;
    Time time;
} XSelectionClearEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *selection* member is set to the selection atom. The *time* member is set to the last change time recorded for the selection. The *window* member is the window that was specified by the current owner (the owner losing the selection) in its *XSetSelectionOwner()* call.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XReparentEvent(), *XResizeRequestEvent()*, *XSelectionEvent()*, *XSelectionRequestEvent()*,
XSetSelectionOwner(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XSelectionEvent – SelectionNotify event structure

STRUCTURES

The structure for SelectionNotify events contains:

```
typedef struct {
    int type;                SelectionNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window requestor;
    Atom selection;
    Atom target;
    Atom property;           atom or None
    Time time;
} XSelectionEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *requestor* member is set to the window associated with the requestor of the selection. The *selection* member is set to the atom that indicates the selection. For example, PRIMARY is used for the primary selection. The *target* member is set to the atom that indicates the converted type. For example, PIXMAP is used for a pixmap. The *property* member is set to the atom that indicates which property the result was stored on. If the conversion failed, the *property* member is set to None. The *time* member is set to the time the conversion took place and can be a timestamp or **CurrentTime**.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XReparentEvent(), *XResizeRequestEvent()*, *XSelectionClearEvent()*,
XSelectionRequestEvent(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XSelectionRequestEvent – SelectionRequest event structure

STRUCTURES

The structure for SelectionRequest events contains:

```
typedef struct {
    int type;                SelectionRequest
    unsigned long serial;    # of last request processed by server
    Bool send_event;        true if this came from a SendEvent request
    Display *display;       Display the event was read from
    Window owner;
    Window requestor;
    Atom selection;
    Atom target;
    Atom property;
    Time time;
} XSelectionRequestEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a **SendEvent** protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *owner* member is set to the window that was specified by the current owner in its **XSetSelectionOwner** call. The *requestor* member is set to the window requesting the selection. The *selection* member is set to the atom that names the selection. For example, PRIMARY is used to indicate the primary selection. The *target* member is set to the atom that indicates the type the selection is desired in. The *property* member can be a property name or None. The *time* member is set to the timestamp or CurrentTime value from the ConvertSelection request.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XReparentEvent(), *XResizeRequestEvent()*, *XSelectionClearEvent()*, *XSelectionEvent()*,
XSetSelectionOwner(), *XUnmapEvent()*, *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XSendEvent(), XDisplayMotionBufferSize(), XGetMotionEvents(), XTimeCoord() – send events and pointer motion history structure

SYNOPSIS

```
Status XSendEvent (Display *display, Window w, Bool propagate,
                  long event_mask, XEvent *event_send)
unsigned long XDisplayMotionBufferSize (Display *display)
XTimeCoord *XGetMotionEvents (Display *display, Window w,
                             Time start, Time stop,
                             int *nevents_return)
```

ARGUMENTS

display Specifies the connection to the X server.

event_mask Specifies the event mask.

event_send Specifies the event that is to be sent.

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

propagate Specifies a Boolean value.

start

stop Specify the time interval in which the events are returned from the motion history buffer. You can pass a timestamp or CurrentTime, PointerWindow, or InputFocus.

w Specifies the window the window the event is to be sent to,.

DESCRIPTION

The *XSendEvent()* function identifies the destination window, determines which clients should receive the specified events, and ignores any active grabs. This function requires you to pass an event mask. For a discussion of the valid event mask names, see section 10.3. This function uses the *w* argument to identify the destination window as follows:

- If *w* is *PointerWindow*, the destination window is the window that contains the pointer.
- If *w* is *InputFocus* and if the focus window contains the pointer, the destination window is the window that contains the pointer; otherwise, the destination window is the focus window.

To determine which clients should receive the specified events, *XSendEvent()* uses the *propagate* argument as follows:

- If *event_mask* is the empty set, the event is sent to the client that created the destination window. If that client no longer exists, no event is sent.
- If *propagate* is *False*, the event is sent to every client selecting on destination any of the event types in the *event_mask* argument.
- If *propagate* is *True* and no clients have selected on destination any of the event types in event-mask, the destination is replaced with the closest ancestor of destination for which some client has selected a type in event-mask and for which no intervening window has that type in its do-not-propagate-mask. If no such window exists or if

the window is an ancestor of the focus window and InputFocus was originally specified as the destination, the event is not sent to any clients. Otherwise, the event is reported to every client selecting on the final destination any of the types specified in *event_mask*.

The event in the **XEvent** structure must be one of the core events or one of the events defined by an extension (or a BadValue error results) so that the X server can correctly byte-swap the contents as necessary. The contents of the event are otherwise unaltered and unchecked by the X server except to force *send_event* to True in the forwarded event and to set the serial number in the event correctly.

XSendEvent() returns zero if the conversion to wire protocol format failed and returns nonzero otherwise. *XSendEvent()* can generate BadValue and BadWindow errors.

The server may retain the recent history of the pointer motion and do so to a finer granularity than is reported by MotionNotify events. The *XGetMotionEvents()* function makes this history available.

The *XGetMotionEvents()* function returns all events in the motion history buffer that fall between the specified *start* and *stop* times, inclusive, and that have coordinates that lie within the specified window (including its borders) at its present placement. If the server does not support motion history, or if the *start* time is later than the *stop* time, or if the *start* time is in the future, no events are returned, and *XGetMotionEvents()* returns NULL. If the time is in the future, it is equivalent to specifying CurrentTime.

XGetMotionEvents() can generate a BadWindow error.

STRUCTURES

The **XTimeCoord** structure contains:

```
typedef struct {
    Time time;
    short x, y;
} XTimeCoord;
```

The *time* member is set to the time, in milliseconds. The *x* and *y* members are set to the coordinates of the pointer and are reported relative to the origin of the specified window.

DIAGNOSTICS

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAnyEvent(), *XIfEvent()*, *XNextEvent()*, *XPutBackEvent()*

Xlib — C Language X Interface

NAME

XSetArcMode(), XSetSubwindowMode(), XSetGraphicsExposure() – GC convenience routines

SYNOPSIS

```
XSetArcMode (Display *display, GC gc, int arc_mode)
XSetSubwindowMode (Display *display, GC gc, int subwindow_mode)
XSetGraphicsExposures (Display *display, GC gc,
                       Bool graphics_exposures)
```

ARGUMENTS

arc_mode Specifies the arc mode. You can pass ArcChord or ArcPieSlice.

display Specifies the connection to the X server.

gc Specifies the GC.

graphics_exposures Specifies a Boolean value that indicates whether you want GraphicsExpose and NoExpose events to be reported when calling *XCopyArea()* and *XCopyPlane()* with this GC.

subwindow_mode Specifies the subwindow mode. You can pass ClipByChildren or IncludeInferiors.

DESCRIPTION

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

XSetArcMode() can generate BadAlloc, BadGC, and BadValue errors.

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

XSetSubwindowMode() can generate BadAlloc, BadGC, and BadValue errors.

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

XSetGraphicsExposures() can generate BadAlloc, BadGC, and BadValue errors.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.

BadGC A value for a GContext argument does not name a defined GContext.

BadValue Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XCopyArea(), *XCreateGC()*, *XQueryBestSize()*, *XSetClipOrigin()*, *XSetFillStyle()*, *XSetFont()*, *XSetLineAttributes()*, *XSetState()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XSetClipOrigin(), XSetClipMask(), XSetClipRectangles() – GC convenience routines

SYNOPSIS

```
XSetClipOrigin (Display *display, GC gc, int clip_x_origin,
               int clip_y_origin)
XSetClipMask (Display *display, GC gc, Pixmap pixmap)
XSetClipRectangles (Display *display, GC gc,
                   int clip_x_origin, int clip_y_origin,
                   XRectangle rectangles[], int n, int ordering)
```

ARGUMENTS

display Specifies the connection to the X server.
clip_x_origin
clip_y_origin Specify the x and y coordinates of the clip-mask origin.
gc Specifies the GC.
n Specifies the number of rectangles.
ordering Specifies the ordering relations on the rectangles. You can pass Unsorted, YSorted, YXSorted, or YXBanded.
pixmap Specifies the pixmap or None .
rectangles Specifies an array of rectangles that define the clip-mask.

DESCRIPTION

The *XSetClipOrigin()* function sets the clip origin in the specified GC. The clip-mask origin is interpreted relative to the origin of whatever destination drawable is specified in the graphics request.

XSetClipOrigin() can generate BadAlloc and BadGC errors.

The *XSetClipMask()* function sets the clip-mask in the specified GC to the specified pixmap. If the clip-mask is set to None , the pixels are always drawn (regardless of the clip-origin).

XSetClipMask() can generate BadAlloc, BadGC, BadMatch, and BadValue errors.

The *XSetClipRectangles()* function changes the clip-mask in the specified GC to the specified list of rectangles and sets the clip origin. The output is clipped to remain contained within the rectangles. The clip-origin is interpreted relative to the origin of whatever destination drawable is specified in a graphics request. The rectangle coordinates are interpreted relative to the clip-origin. The rectangles should be nonintersecting, or the graphics results will be undefined. Note that the list of rectangles can be empty, which effectively disables output. This is the opposite of passing None as the clip-mask in *XCreateGC()*, *XChangeGC()*, and *XSetClipMask()*.

If known by the client, ordering relations on the rectangles can be specified with the ordering argument. This may provide faster operation by the server. If an incorrect ordering is specified, the X server may generate a BadMatch error, but it is not required to do so. If no error is generated, the graphics results are undefined. Unsorted means the rectangles are in arbitrary order. YSorted means that the rectangles are nondecreasing in their Y origin. YXSorted additionally constrains YSorted order in that all rectangles with an equal Y origin are nondecreasing in their X origin. YXBanded

additionally constrains YXSorted by requiring that, for every possible Y scanline, all rectangles that include that scanline have an identical Y origins and Y extents.

XSetClipRectangles() can generate BadAlloc , BadGC , BadMatch , and BadValue errors.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadGC A value for a GContext argument does not name a defined GContext.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XCreateGC(), *XDrawRectangle()*, *XQueryBestSize()*, *XSetArcMode()*, *XSetFillStyle()*, *XSetFont()*, *XSetLineAttributes()*, *XSetState()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XSetCloseDownMode(), XKillClient() – control clients

SYNOPSIS

```
XSetCloseDownMode (Display *display, int close_mode)
XKillClient (Display *display, XID resource)
```

ARGUMENTS

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

display Specifies the connection to the X server.

resource Specifies any resource associated with the client that you want to destroy or AllTemporary .

DESCRIPTION

The *XSetCloseDownMode()* defines what will happen to the client's resources at connection close. A connection starts in DestroyAll mode. For information on what happens to the client's resources when the *close_mode* argument is RetainPermanent or RetainTemporary, see section 2.6.

XSetCloseDownMode() can generate a BadValue error.

The *XKillClient()* function forces a close-down of the client that created the resource if a valid resource is specified. If the client has already terminated in either RetainPermanent or RetainTemporary mode, all of the client's resources are destroyed. If *AllTemporary()* is specified, the resources of all clients that have terminated in *RetainTemporary()* are destroyed (see section 2.5). This permits implementation of window manager facilities that aid debugging. A client can set its close-down mode to RetainTemporary. If the client then crashes, its windows would not be destroyed. The programmer can then inspect the application's window tree and use the window manager to destroy the zombie windows.

XKillClient() can generate a BadValue error.

DIAGNOSTICS

BadValue
Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

Xlib — C Language X Interface

NAME

XSetCommand(), XGetCommand() – set or read a window's WM_COMMAND property

SYNOPSIS

```
XSetCommand (Display *display, Window w, char **argv, int argc)
Status XGetCommand (Display *display, Window w,
                   char ***argv_return, int *argc_return)
```

ARGUMENTS

argc Specifies the number of arguments.
argc_return Returns the number of arguments returned.
argv Specifies the application's argument list.
argv_return Returns the application's argument list.
display Specifies the connection to the X server.
w Specifies the window.

DESCRIPTION

The *XSetCommand()* function sets the command and arguments used to invoke the application. (Typically, *argv* is the *argv* array of your main program.) If the strings are not in the Host Portable Character Encoding the result is implementation dependent.

XSetCommand() can generate BadAlloc and BadWindow errors.

The *XGetCommand()* function reads the WM_COMMAND property from the specified window and returns a string list. If the WM_COMMAND property exists, it is of type STRING and format 8. If sufficient memory can be allocated to contain the string list, *XGetCommand()* fills in the *argv_return* and *argc_return* arguments and returns a nonzero status. Otherwise, it returns a zero status. If the data returned by the server is in the Latin Portable Character Encoding, then the returned strings are in the Host Portable Character Encoding. Otherwise, the result is implementation dependent. To free the memory allocated to the string list, use *XFreeStringList()*.

PROPERTIES

WM_COMMAND
 The command and arguments, null-separated, used to invoke the application.

DIAGNOSTICS

BadAlloc
 The server failed to allocate the requested resource or server memory.
 BadWindow
 A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), *XAllocIconSize()*, *XAllocSizeHints()*, *XAllocWMHints()*,
XSetTransientForHint(), *XSetTextProperty()*, *XSetWMClientMachine()*,
XSetWMColormapWindows(), *XSetWMIconName()*, *XSetWMName()*,
XSetWMProperties(), *XSetWMProtocols()*, *XStringListToTextProperty()*

Xlib — C Language X Interface

NAME

XSetErrorHandler(), XGetErrorText(), XDisplayName(), XSetIOErrorHandler(),
XGetErrorDatabaseText() – default error handlers

SYNOPSIS

```
int XGetErrorText (Display *display, int code,
                  char *buffer_return, int length)
char *XDisplayName (char *string)
int XSetErrorHandler (int (* handler)(Display *))
XGetErrorDatabaseText (Display *display, char *name,
                      char *message, char *default_string,
                      char *buffer_return, int length)
```

ARGUMENTS

buffer_return Returns the error description.

code Specifies the error code for which you want to obtain a description.

default_string Specifies the default error message if none is found in the database.

display Specifies the connection to the X server.

handler Specifies the program's supplied error handler.

length Specifies the size of the buffer.

message Specifies the type of the error message.

name Specifies the name of the application.

string Specifies the character string.

DESCRIPTION

Xlib generally calls the program's supplied error handler whenever an error is received. It is not called on BadName errors from OpenFont, LookupColor, or AllocNamedColor protocol requests or on BadFont errors from a QueryFont protocol request. These errors generally are reflected back to the program through the procedural interface. Because this condition is not assumed to be fatal, it is acceptable for your error handler to return. However, the error handler should not call any functions (directly or indirectly) on the display that will generate protocol requests or that will look for input events. The previous error handler is returned.

The *XGetErrorText()* function copies a null-terminated string describing the specified error code into the specified buffer. The returned text is in the encoding of the current locale. It is recommended that you use this function to obtain an error description because extensions to Xlib may define their own error codes and error strings.

The *XDisplayName()* function returns the name of the display that *XOpenDisplay()* would attempt to use. If a NULL string is specified, *XDisplayName()* looks in the environment for the display and returns the display name that *XOpenDisplay()* would attempt to use. This makes it easier to report to the user precisely which display the program attempted to open when the initial connection attempt failed.

The *XSetIOErrorHandler()* sets the fatal I/O error handler. Xlib calls the program's supplied error handler if any sort of system call error occurs (for example, the connection to the server was lost). This is assumed to be a fatal condition, and the called routine should not return. If the I/O error handler does return, the client process exits.

Note that the previous error handler is returned.

The *XGetErrorDatabaseText()* function returns a null-terminated message (or the default message) from the error message database. Xlib uses this function internally to look up its error messages. The *default_string* is assumed to be in the encoding of the current locale. The *buffer_return* text is in the encoding of the current locale.

The *name* argument should generally be the name of your application. The *message* argument should indicate which type of error message you want. If the *name* and *message* are not in the Host Portable Character Encoding the result is implementation dependent. Xlib uses three predefined “application names” to report errors (uppercase and lowercase matter):

XProtoError

The protocol error number is used as a string for the message argument.

XlibMessage

These are the message strings that are used internally by the library.

XRequest

For a core protocol request, the major request protocol number is used for the message argument. For an extension request, the extension name (as given by *InitExtension()*) followed by a period (.) and the minor request protocol number is used for the message argument. If no string is found in the error database, the *default_string* is returned to the buffer argument.

SEE ALSO

XOpenDisplay(), *XSynchronize()*

Xlib — C Language X Interface

NAME

XSetFillStyle(), XSetFillRule() – GC convenience routines

SYNOPSIS

```
XSetFillStyle (Display *display, GC gc, int fill_style)
XSetFillRule (Display *display, GC gc, int fill_rule)
```

ARGUMENTS

display Specifies the connection to the X server.

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

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

gc Specifies the GC.

DESCRIPTION

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

XSetFillStyle() can generate BadAlloc, BadGC, and BadValue errors.

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

XSetFillRule() can generate BadAlloc, BadGC, and BadValue errors.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.

BadGC A value for a GContext argument does not name a defined GContext.

BadValue Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XCreateGC(), *XQueryBestSize()*, *XSetArcMode()*, *XSetClipOrigin()*, *XSetFont()*, *XSetLineAttributes()*, *XSetState()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XSetFont() – GC convenience routines

SYNOPSIS

XSetFont (Display **display*, GC *gc*, Font *font*)

ARGUMENTS

display Specifies the connection to the X server.
font Specifies the font.
gc Specifies the GC.

DESCRIPTION

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

XSetFont() can generate BadAlloc, BadFont, and BadGC errors.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.
BadFont A value for a Font or GContext argument does not name a defined Font.
BadGC A value for a GContext argument does not name a defined GContext.

SEE ALSO

XCreateGC(), *XQueryBestSize()*, *XSetArcMode()*, *XSetClipOrigin()*, *XSetFillStyle()*,
XSetLineAttributes(), *XSetState()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XSetFontPath(), XGetFontPath(), XFreeFontPath() – set, get, or free the font search path

SYNOPSIS

```
XSetFontPath (Display *display, char **directories, int ndirs)
char **XGetFontPath (Display *display, int *npaths_return)
XFreeFontPath (char **list)
```

ARGUMENTS

directories

Specifies the directory path used to look for a font. Setting the path to the empty list restores the default path defined for the X server.

display Specifies the connection to the X server.

list Specifies the array of strings you want to free.

ndirs Specifies the number of directories in the path.

npaths_return

Returns the number of strings in the font path array.

DESCRIPTION

The *XSetFontPath()* function defines the directory search path for font lookup. There is only one search path per X server, not one per client. The encoding and interpretation of the strings is implementation dependent, but typically they specify directories or font servers to be searched in the order listed. An X server is permitted to cache font information internally, for example, it might cache an entire font from a file and not check on subsequent opens of that font to see if the underlying font file has changed. However, when the font path is changed the X server is guaranteed to flush all cached information about fonts for which there currently are no explicit resource IDs allocated. The meaning of an error from this request is implementation dependent.

XSetFontPath() can generate a BadValue error.

The *XGetFontPath()* function allocates and returns an array of strings containing the search path. The contents of these strings are implementation dependent and are not intended to be interpreted by client applications. When it is no longer needed, the data in the font path should be freed by using *XFreeFontPath()*.

The *XFreeFontPath()* function frees the data allocated by *XGetFontPath()*.

DIAGNOSTICS

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XListFonts(), *XLoadFont()*

Xlib — C Language X Interface

NAME

XSetICFocus(), XUnsetICFocus() – set and unset input context focus

SYNOPSIS

```
void XSetICFocus (XIC ic)  
void XUnsetICFocus (XIC ic)
```

ARGUMENTS

ic Specifies the input context.

DESCRIPTION

The *XSetICFocus()* function allows a client to notify an input method that the focus window attached to the specified input context has received keyboard focus. The input method should take action to provide appropriate feedback. Complete feedback specification is a matter of user interface policy.

The *XUnsetICFocus()* function allows a client to notify an input method that the specified input context has lost the keyboard focus and that no more input is expected on the focus window attached to that input context. The input method should take action to provide appropriate feedback. Complete feedback specification is a matter of user interface policy.

SEE ALSO

XCreateIC(), *XOpenIM()*, *XSetICValues()*, *XmbResetIC()*

Xlib — C Language X Interface

NAME

XSetICValues(), XGetICValues() – set and obtain XIC values

SYNOPSIS

```
char * XSetICValues (XIC ic, ...)  
char * XGetICValues (XIC ic, ...)
```

ARGUMENTS

ic Specifies the input context.
... Specifies the variable length argument list to set or get XIC values.

DESCRIPTION

The *XSetICValues()* function returns NULL if no error occurred; otherwise, it returns the name of the first argument that could not be set. An argument could be not set for any of the following reasons:

- A read-only argument was set (for example, XNFilterEvents).
- The argument name is not recognized.
- The input method encountered an input method implementation dependent error.

Each value to be set must be an appropriate datum, matching the data type imposed by the semantics of the argument.

The *XSetICValues()* can generate BadAtom, BadColor, BadCursor, BadPixmap, and BadWindow errors.

The *XGetICValues()* function returns NULL if no error occurred; otherwise, it returns the name of the first argument that could not be obtained. An argument could be not obtained for any of the following reasons:

- The argument name is not recognized.
- The input method encountered an implementation dependent error.

Each argument value (following a name) must point to a location where the value is to be stored. *XGetICValues()* allocates memory to store the values, and client is responsible for freeing each value by calling *XFree()*.

DIAGNOSTICS

BadAtom

A value for an Atom argument does not name a defined Atom.

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadCursor

A value for a Cursor argument does not name a defined Cursor.

BadPixmap

A value for a Pixmap argument does not name a defined Pixmap.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XCreateIC(), *XOpenIM()*, *XSetICFocus()*, *XmbResetIC()*

Xlib — C Language X Interface

NAME

XSetInputFocus(), XGetInputFocus() – control input focus

SYNOPSIS

```
XSetInputFocus (Display *display, Window focus, int revert_to,
                Time time)
XGetInputFocus (Display *display, Window *focus_return,
                int *revert_to_return)
```

ARGUMENTS

display Specifies the connection to the X server.

focus Specifies the window, PointerRoot, or None.

focus_return
Returns the focus window, PointerRoot, or None.

revert_to
Specifies where the input focus reverts to if the window becomes not viewable. You can pass RevertToParent, RevertToPointerRoot, or RevertToNone.

revert_to_return
Returns the current focus state (RevertToParent, RevertToPointerRoot, or RevertToNone).

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

DESCRIPTION

The *XSetInputFocus()* function changes the input focus and the last-focus-change time. It has no effect if the specified time is earlier than the current last-focus-change time or is later than the current X server time. Otherwise, the last-focus-change time is set to the specified time (CurrentTime is replaced by the current X server time). *XSetInputFocus()* causes the X server to generate FocusIn and FocusOut events.

Depending on the focus argument, the following occurs:

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

The specified focus window must be viewable at the time *XSetInputFocus()* is called, or a BadMatch error results. If the focus window later becomes not viewable, the X server evaluates the *revert_to* argument to determine the new focus window as follows:

- If *revert_to* is RevertToParent, the focus reverts to the parent (or the closest viewable ancestor), and the new *revert_to* value is taken to be RevertToNone.
- If *revert_to* is RevertToPointerRoot or RevertToNone, the focus reverts to PointerRoot or None, respectively. When the focus reverts, the X server generates FocusIn and FocusOut events, but the last-focus-change time is not affected.

XSetInputFocus() can generate BadMatch, BadValue, and BadWindow errors.

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

DIAGNOSTICS**BadValue**

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XWarpPointer()

Xlib — C Language X Interface

NAME

XSetLineAttributes(), XSetDashes() – GC convenience routines

SYNOPSIS

```
XSetLineAttributes (Display *display, GC gc,
                   unsigned int line_width, int line_style,
                   int cap_style, int join_style)
XSetDashes (Display *display, GC gc, int dash_offset,
           char dash_list[], int n)
```

ARGUMENTS

cap_style Specifies the line-style and cap-style you want to set for the specified GC. You can pass CapNotLast, CapButt, CapRound, or CapProjecting.

dash_list Specifies the dash-list for the dashed line-style you want to set for the specified GC.

dash_offset Specifies the phase of the pattern for the dashed line-style you want to set for the specified GC.

display Specifies the connection to the X server.

gc Specifies the GC.

join_style Specifies the line join-style you want to set for the specified GC. You can pass JoinMiter, JoinRound, or JoinBevel.

line_style Specifies the line-style you want to set for the specified GC. You can pass LineSolid, LineOnOffDash, or LineDoubleDash.

line_width Specifies the line-width you want to set for the specified GC.

n Specifies the number of elements in *dash_list*.

DESCRIPTION

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

XSetLineAttributes() can generate BadAlloc, BadGC, and BadValue errors.

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

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

The unit of measure for dashes is the same for the ordinary coordinate system. Ideally, a dash length is measured along the slope of the line, but implementations are only required to match this ideal for horizontal and vertical lines. Failing the ideal semantics,

it is suggested that the length be measured along the major axis of the line. The major axis is defined as the x axis for lines drawn at an angle of between -45 and +45 degrees or between 135 and 225 degrees from the x axis. For all other lines, the major axis is the y axis.

XSetDashes() can generate BadAlloc, BadGC, and BadValue errors.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadGC A value for a GContext argument does not name a defined GContext.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XCreateGC(), *XQueryBestSize()*, *XSetArcMode()*, *XSetClipOrigin()*, *XSetFillStyle()*, *XSetFont()*, *XSetState()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XSetPointerMapping(), XGetPointerMapping() – manipulate pointer settings

SYNOPSIS

```
int XSetPointerMapping (Display *display, unsigned char map[],
                        int nmap)
int XGetPointerMapping (Display *display,
                        unsigned char map_return[],
                        int nmap)
```

ARGUMENTS

display Specifies the connection to the X server.
map Specifies the mapping list.
map_return Returns the mapping list.
nmap Specifies the number of items in the mapping list.

DESCRIPTION

The *XSetPointerMapping()* function sets the mapping of the pointer. If it succeeds, the X server generates a MappingNotify event, and *XSetPointerMapping()* returns MappingSuccess. Element *map*[*i*] defines the logical button number for the physical button *i*+1. The length of the list must be the same as *XGetPointerMapping()* would return, or a BadValue error results. A zero element disables a button, and elements are not restricted in value by the number of physical buttons. However, no two elements can have the same nonzero value, or a BadValue error results. If any of the buttons to be altered are logically in the down state, *XSetPointerMapping()* returns MappingBusy, and the mapping is not changed.

XSetPointerMapping() can generate a BadValue error.

The *XGetPointerMapping()* function returns the current mapping of the pointer. Pointer buttons are numbered starting from one. *XGetPointerMapping()* returns the number of physical buttons actually on the pointer. The nominal mapping for a pointer is *map*[*i*]=*i*+1. The *nmap* argument specifies the length of the array where the pointer mapping is returned, and only the first *nmap* elements are returned in *map_return*.

DIAGNOSTICS

BadValue
 Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XChangeKeyboardControl(), *XChangeKeyboardMapping()*

Xlib — C Language X Interface

NAME

XSetScreenSaver(), XForceScreenSaver(), XActivateScreenSaver(), XResetScreenSaver(), XGetScreenSaver() – manipulate the screen saver

SYNOPSIS

```
XSetScreenSaver (Display *display, int timeout, int interval,
                 int prefer_blanking, int allow_exposures)
XForceScreenSaver (Display *display, int mode)
XActivateScreenSaver (Display *display)
XResetScreenSaver (Display *display)
XGetScreenSaver (Display *display, int *timeout_return,
                 int *interval_return, int *prefer_blanking_return,
                 int *allow_exposures_return)
```

ARGUMENTS

allow_exposures
Specifies the screen save control values. You can pass DontAllowExposures, AllowExposures, or DefaultExposures.

allow_exposures_return
Returns the current screen save control value (DontAllowExposures, AllowExposures, or DefaultExposures).

display
Specifies the connection to the X server.

interval
Specifies the interval, in seconds, between screen saver alterations.

interval_return
Returns the interval between screen saver invocations.

mode
Specifies the mode that is to be applied. You can pass ScreenSaverActive or ScreenSaverReset.

prefer_blanking
Specifies how to enable screen blanking. You can pass DontPreferBlanking, PreferBlanking, or DefaultBlanking.

prefer_blanking_return
Returns the current screen blanking preference (DontPreferBlanking, PreferBlanking, or DefaultBlanking).

timeout
Specifies the timeout, in seconds, until the screen saver turns on.

timeout_return
Returns the timeout, in seconds, until the screen saver turns on.

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 nonzero, *XSetScreenSaver()* enables the screen saver. An *interval* of 0 disables the random-pattern motion. If no input from devices (keyboard, mouse, and so on) is generated for the specified number of *timeout* seconds once the screen saver is enabled, the screen saver is activated.

For each screen, if blanking is preferred and the hardware supports video blanking, the screen simply goes blank. Otherwise, if either exposures are allowed or the screen can be regenerated without sending Expose events to clients, the screen is tiled with the root window background tile randomly re-originated 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-originated periodically.

`XSetScreenSaver()` can generate a `BadValue` error.

If the specified mode is `ScreenSaverActive` and the screen saver currently is deactivated, `XForceScreenSaver()` activates the screen saver even if the screen saver had been disabled with a *timeout* of zero. If the specified mode is `ScreenSaverReset` and the screen saver currently is enabled, `XForceScreenSaver()` deactivates the screen saver if it was activated, and the activation timer is reset to its initial state (as if device input had been received).

`XForceScreenSaver()` can generate a `BadValue` error.

The `XActivateScreenSaver()` function activates the screen saver.

The `XResetScreenSaver()` function resets the screen saver.

The `XGetScreenSaver()` function gets the current screen saver values.

DIAGNOSTICS

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

Xlib — C Language X Interface

NAME

XSetSelectionOwner(), XGetSelectionOwner(), XConvertSelection() – manipulate window selection

SYNOPSIS

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

ARGUMENTS

display Specifies the connection to the X server.
owner Specifies the owner of the specified selection atom. You can pass a window or None.
property Specifies the property name. You also can pass None.
requestor Specifies the requestor.
selection Specifies the selection atom.
target Specifies the target atom.
time Specifies the time. You can pass either a timestamp or CurrentTime.

DESCRIPTION

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

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

XGetSelectionOwner() returns the owner window, which is reported in SelectionRequest and SelectionClear events. Selections are global to the X server.

XSetSelectionOwner() can generate BadAtom and BadWindow errors.

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

XGetSelectionOwner() can generate a BadAtom error.

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

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

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

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

XConvertSelection() can generate BadAtom and BadWindow errors.

DIAGNOSTICS

BadAtom

A value for an Atom argument does not name a defined Atom.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

Xlib — C Language X Interface

NAME

XSetState(), XSetFunction(), XSetPlanemask(), XSetForeground(), XSetBackground() – GC convenience routines

SYNOPSIS

```
XSetState (Display *display, GC gc,
          unsigned long foreground,
          unsigned long background,
          int function, unsigned long plane_mask)
XSetFunction (Display *display, GC gc, int function)
XSetPlaneMask (Display *display, GC gc,
              unsigned long plane_mask)
XSetForeground (Display *display, GC gc,
              unsigned long foreground)
XSetBackground (Display *display, GC gc,
              unsigned long background)
```

ARGUMENTS

background Specifies the background you want to set for the specified GC.
display Specifies the connection to the X server.
foreground Specifies the foreground you want to set for the specified GC.
function Specifies the function you want to set for the specified GC.
gc Specifies the GC.
plane_mask Specifies the plane mask.

DESCRIPTION

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

XSetState() can generate BadAlloc, BadGC, and BadValue errors.

XSetFunction() sets a specified value in the specified GC.

XSetFunction() can generate BadAlloc, BadGC, and BadValue errors.

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

XSetPlaneMask() can generate BadAlloc and BadGC errors.

The *XSetForeground()* function sets the foreground in the specified GC.

XSetForeground() can generate BadAlloc and BadGC errors.

The *XSetBackground()* function sets the background in the specified GC.

XSetBackground() can generate BadAlloc and BadGC errors.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.
 BadGC A value for a GContext argument does not name a defined GContext.
 BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XCreateGC(), *XQueryBestSize()*, *XSetArcMode()*, *XSetClipOrigin()*, *XSetFillStyle()*,
XSetFont(), *XSetLineAttributes()*, *XSetTile()*

Xlib — C Language X Interface

NAME

XSetTextProperty(), XGetTextProperty() – set and read text properties

SYNOPSIS

```
void XSetTextProperty (Display *display, Window w,
                      XTextProperty *text_prop, Atom property)
Status XGetTextProperty (Display *display, Window w,
                        XTextProperty *text_prop_return, Atom property)
```

ARGUMENTS

display Specifies the connection to the X server.
property Specifies the property name.
text_prop
 Specifies the **XTextProperty** structure to be used.
text_prop_return
 Returns the **XTextProperty** structure.

DESCRIPTION

The *XSetTextProperty()* function replaces the existing specified property for the named window with the data, type, format, and number of items determined by the value field, the encoding field, the format field, and the nitens field, respectively, of the specified **XTextProperty** structure. If the property does not already exist, *XSetTextProperty()* sets it for the specified window.

XSetTextProperty() can generate BadAlloc, BadAtom, BadValue, and BadWindow errors.

The *XGetTextProperty()* function reads the specified property from the window and stores the data in the returned **XTextProperty** structure. It stores the data in the value field, the type of the data in the encoding field, the format of the data in the format field, and the number of items of data in the nitens field. An extra byte containing null (which is not included in the nitens member) is stored at the end of the value field of *text_prop_return*. The particular interpretation of the property's encoding and data as "text" is left to the calling application. If the specified property does not exist on the window, *XGetTextProperty()* sets the *value* field to NULL, the *encoding* field to None, the *format* field to zero, and the *nitens* field to zero.

If it was able to read and store the data in the **XTextProperty** structure, *XGetTextProperty()* returns a nonzero status; otherwise, it returns a zero status.

XGetTextProperty() can generate BadAtom and BadWindow errors.

PROPERTIES

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

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadAtom

A value for an Atom argument does not name a defined Atom.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), XAllocIconSize(), XAllocSizeHints(), XAllocWMHints(), XSetCommand(), XSetTransientForHint(), XSetWMClientMachine(), XSetWMColormapWindows(), XSetWMIconName(), XSetWMName(), XSetWMProperties(), XSetWMProtocols(), XStringListToTextProperty()

Xlib — C Language X Interface

NAME

XSetTile(), XSetStipple(), XSetTSOrigin() – GC convenience routines

SYNOPSIS

```
XSetTile (Display *display, GC gc, Pixmap tile)
XSetStipple (Display *display, GC gc, Pixmap stipple)
XSetTSOrigin (Display *display, GC gc, int ts_x_origin,
              int ts_y_origin)
```

ARGUMENTS

display Specifies the connection to the X server.
gc Specifies the GC.
stipple Specifies the stipple you want to set for the specified GC.
tile Specifies the fill tile you want to set for the specified GC.
ts_x_origin
ts_y_origin Specify the x and y coordinates of the tile and stipple origin.

DESCRIPTION

The *XSetTile()* function sets the fill tile in the specified GC. The tile and GC must have the same depth, or a BadMatch error results.

XSetTile() can generate BadAlloc, BadGC, BadMatch, and BadPixmap errors.

The *XSetStipple()* function sets the stipple in the specified GC. The stipple must have a depth of one, or a BadMatch error results.

XSetStipple() can generate BadAlloc, BadGC, BadMatch, and BadPixmap errors.

The *XSetTSOrigin()* function sets the tile / stipple origin in the specified GC. When graphics requests call for tiling or stippling, the parent's origin will be interpreted relative to whatever destination drawable is specified in the graphics request.

XSetTSOrigin() can generate BadAlloc and BadGC errors.

DIAGNOSTICS

BadAlloc The server failed to allocate the requested resource or server memory.
 BadGC A value for a GContext argument does not name a defined GContext.
 BadMatch Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.
 BadPixmap A value for a Pixmap argument does not name a defined Pixmap.

SEE ALSO

XCreateGC(), *XQueryBestSize()*, *XSetArcMode()*, *XSetClipOrigin()*, *XSetFillStyle()*, *XSetFont()*, *XSetLineAttributes()*, *XSetState()*

Xlib — C Language X Interface

NAME

XSetTransientForHint(), XGetTransientForHint() – set or read a window's WM_TRANSIENT_FOR property

SYNOPSIS

```
XSetTransientForHint (Display *display, Window w,
                     Window prop_window)
Status XGetTransientForHint (Display *display, Window w,
                             Window *prop_window_return)
```

ARGUMENTS

display Specifies the connection to the X server.
w Specifies the window.
prop_window
 Specifies the window that the WM_TRANSIENT_FOR property is to be set to.
prop_window_return
 Returns the WM_TRANSIENT_FOR property of the specified window.

DESCRIPTION

The *XSetTransientForHint()* function sets the WM_TRANSIENT_FOR property of the specified window to the specified *prop_window*.

XSetTransientForHint() can generate BadAlloc and BadWindow errors.

The *XGetTransientForHint()* function returns the WM_TRANSIENT_FOR property for the specified window. It returns nonzero status on success; otherwise it returns a zero status.

XGetTransientForHint() can generate a BadWindow error.

PROPERTIES

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

DIAGNOSTICS

BadAlloc
 The server failed to allocate the requested resource or server memory.
 BadWindow
 A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), *XAllocIconSize()*, *XAllocSizeHints()*, *XAllocWMHints()*,
XSetCommand(), *XSetTextProperty()*, *XSetWMClientMachine()*,
XSetWMColormapWindows(), *XSetWMIconName()*, *XSetWMName()*,
XSetWMProperties(), *XSetWMProtocols()*, *XStringListToTextProperty()*

Xlib — C Language X Interface

NAME

`XSetWMClientMachine()`, `XGetWMClientMachine()` – set or read a window's `WM_CLIENT_MACHINE` property

SYNOPSIS

```
void XSetWMClientMachine (Display *display, Window w,
                          XTextProperty *text_prop)
Status XGetWMClientMachine (Display *display, Window w,
                            XTextProperty *text_prop_return)
```

ARGUMENTS

display Specifies the connection to the X server.
text_prop Specifies the **XTextProperty** structure to be used.
text_prop_return Returns the **XTextProperty** structure.
w Specifies the window.

DESCRIPTION

The **XSetWMClientMachine** convenience function calls `XSetTextProperty()` to set the `WM_CLIENT_MACHINE` property.

The `XGetWMClientMachine()` convenience function performs an `XGetTextProperty()` on the `WM_CLIENT_MACHINE` property. It returns nonzero status on success; otherwise it returns a zero status.

PROPERTIES

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

SEE ALSO

`XAllocClassHint()`, `XAllocIconSize()`, `XAllocSizeHints()`, `XAllocWMHints()`,
`XSetCommand()`, `XSetTransientForHint()`, `XSetTextProperty()`,
`XSetWMColormapWindows()`, `XSetWMIconName()`, `XSetWMName()`,
`XSetWMProperties()`, `XSetWMProtocols()`, `XStringListToTextProperty()`
Xlib — C Language X Interface

NAME

XSetWMColormapWindows(), XGetWMColormapWindows() – set or read a window's WM_COLORMAP_WINDOWS property

SYNOPSIS

```
Status XSetWMColormapWindows (Display *display, Window w,
                               Window *colormap_windows, int count)
Status XGetWMColormapWindows (Display *display, Window w,
                               Window **colormap_windows_return,
                               int *count_return)
```

ARGUMENTS

display Specifies the connection to the X server.
colormap_windows Specifies the list of windows.
colormap_windows_return Returns the list of windows.
count Specifies the number of windows in the list.
count_return Returns the number of windows in the list.
w Specifies the window.

DESCRIPTION

The *XSetWMColormapWindows()* function replaces the WM_COLORMAP_WINDOWS property on the specified window with the list of windows specified by the *colormap_windows* argument. If the property does not already exist, *XSetWMColormapWindows()* sets the WM_COLORMAP_WINDOWS property on the specified window to the list of windows specified by the *colormap_windows* argument. The property is stored with a type of WINDOW and a format of 32. If it cannot intern the WM_COLORMAP_WINDOWS atom, *XSetWMColormapWindows()* returns a zero status. Otherwise, it returns a nonzero status.

XSetWMColormapWindows() can generate BadAlloc and BadWindow errors.

The *XGetWMColormapWindows()* function returns the list of window identifiers stored in the WM_COLORMAP_WINDOWS property on the specified window. These identifiers indicate the colormaps that the window manager may need to install for this window. If the property exists, is of type WINDOW, is of format 32, and the atom WM_COLORMAP_WINDOWS can be interned, *XGetWMColormapWindows()* sets the *windows_return* argument to a list of window identifiers, sets the *count_return* argument to the number of elements in the list, and returns a nonzero status. Otherwise, it sets neither of the return arguments and returns a zero status. To release the list of window identifiers, use *XFree()*.

XGetWMColormapWindows() can generate a BadWindow error.

PROPERTIES

WM_COLORMAP_WINDOWS

The list of window IDs that may need a different colormap than that of their top-level window.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), XAllocIconSize(), XAllocSizeHints(), XAllocWMHints(), XFree(), XSetCommand(), XSetTransientForHint(), XSetTextProperty(), XSetWMClientMachine(), XSetWMIconName(), XSetWMName(), XSetWMProperties(), XSetWMProtocols(), XStringListToTextProperty()

Xlib — C Language X Interface

NAME

XSetWMIconName(), XGetWMIconName(), XSetIconName(), XGetIconName() – set or read a window's WM_ICON_NAME property

SYNOPSIS

```
void XSetWMIconName (Display *display, Window w,
                    XTextProperty *text_prop)
Status XGetWMIconName (Display *display, Window w,
                     XTextProperty *text_prop_return)
XSetIconName (Display *display, Window w, char *icon_name)
Status XGetIconName (Display *display, Window w,
                   char **icon_name_return)
```

ARGUMENTS

display Specifies the connection to the X server.

icon_name Specifies the icon name, which should be a null-terminated string.

icon_name_return Returns the window's icon name, which is a null-terminated string.

text_prop Specifies the **XTextProperty** structure to be used.

text_prop_return Returns the **XTextProperty** structure.

w Specifies the window.

DESCRIPTION

The *XSetWMIconName()* convenience function calls *XSetTextProperty()* to set the WM_ICON_NAME property.

The *XGetWMIconName()* convenience function calls *XGetTextProperty()* to obtain the WM_ICON_NAME property. It returns nonzero status on success; otherwise it returns a zero status.

The *XSetIconName()* function sets the name to be displayed in a window's icon.

XSetIconName() can generate BadAlloc and BadWindow errors.

The *XGetIconName()* function returns the name to be displayed in the specified window's icon. If it succeeds, it returns nonzero; otherwise, if no icon name has been set for the window, it returns zero. If you never assigned a name to the window, *XGetIconName()* sets *icon_name_return* to NULL. If the data returned by the server is in the Latin Portable Character Encoding, then the returned string is in the Host Portable Character Encoding. Otherwise, the result is implementation dependent. When finished with it, a client must free the icon name string using *XFree()*.

XGetIconName() can generate a BadWindow error.

PROPERTIES

WM_ICON_NAME
The name to be used in an icon.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), *XAllocIconSize()*, *XAllocSizeHints()*, *XAllocWMHints()*, *XFree()*,
XSetCommand(), *XSetTransientForHint()*, *XSetTextProperty()*, *XSetWMClientMachine()*,
XSetWMColormapWindows(), *XSetWMName()*, *XSetWMProperties()*,
XSetWMProtocols(), *XStringListToTextProperty()*

Xlib — C Language X Interface

NAME

XSetWMName(), XGetWMName(), XStoreName(), XFetchName() – set or read a window's WM_NAME property

SYNOPSIS

```
void XSetWMName (Display *display, Window w,
                XTextProperty *text_prop)
Status XGetWMName (Display *display, Window w,
                  XTextProperty *text_prop_return)
XStoreName (Display *display, Window w, char *window_name)
Status XFetchName (Display *display, Window w,
                  char **window_name_return)
```

ARGUMENTS

display Specifies the connection to the X server.

text_prop Specifies the **XTextProperty** structure to be used.

text_prop_return Returns the **XTextProperty** structure.

w Specifies the window.

window_name Specifies the window name, which should be a null-terminated string.

window_name_return Returns the window name, which is a null-terminated string.

DESCRIPTION

The *XSetWMName()* convenience function calls *XSetTextProperty()* to set the WM_NAME property.

The *XGetWMName()* convenience function calls *XGetTextProperty()* to obtain the WM_NAME property. It returns nonzero status on success; otherwise it returns a zero status.

The *XStoreName()* function assigns the name passed to *window_name* to the specified window. A window manager can display the window name in some prominent place, such as the title bar, to allow users to identify windows easily. Some window managers may display a window's name in the window's icon, although they are encouraged to use the window's icon name if one is provided by the application. If the string is not in the Host Portable Character Encoding the result is implementation dependent.

XStoreName() can generate BadAlloc and BadWindow errors.

The *XFetchName()* function returns the name of the specified window. If it succeeds, it returns nonzero; otherwise, no name has been set for the window, and it returns zero. If the WM_NAME property has not been set for this window, *XFetchName()* sets *window_name_return* to NULL. If the data returned by the server is in the Latin Portable Character Encoding, then the returned string is in the Host Portable Character Encoding. Otherwise, the result is implementation dependent. When finished with it, a client must free the window name string using *XFree()*.

XFetchName() can generate a BadWindow error.

PROPERTIES

WM_NAME

The name of the application.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), XAllocIconSize(), XAllocSizeHints(), XAllocWMHints(), XFree(), XSetCommand(), XSetTransientForHint(), XSetTextProperty(), XSetWMClientMachine(), XSetWMColormapWindows(), XSetWMIconName(), XSetWMProperties(), XSetWMProtocols(), XStringListToTextProperty()

Xlib — C Language X Interface

NAME

XSetWMProperties(), XmbSetWMProperties() – set standard window properties

SYNOPSIS

```
void XSetWMProperties (Display *display, Window w,
                     XTextProperty *window_name,
                     XTextProperty *icon_name,
                     char **argv, int argc,
                     XSizeHints *normal_hints,
                     XWMHints *wm_hints,
                     XClassHint *class_hints)
void XmbSetWMProperties (Display *display, Window w,
                      char *window_name, char *icon_name,
                      char *argv[], int argc,
                      XSizeHints *normal_hints,
                      XWMHints *wm_hints,
                      XClassHint *class_hints)
```

ARGUMENTS

argc Specifies the number of arguments.

argv Specifies the application's argument list.

class_hints Specifies the **XClassHint** structure to be used.

display Specifies the connection to the X server.

icon_name Specifies the icon name, which should be a null-terminated string.

normal_hints Specifies the size hints for the window in its normal state.

w Specifies the window.

window_name Specifies the window name, which should be a null-terminated string.

wm_hints Specifies the **XWMHints** structure to be used.

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 *window_name* argument is non-NULL, *XSetWMProperties()* calls *XSetWMName()*, which in turn, sets the WM_NAME property (see section 14.1.4). If the *icon_name* argument is non-NULL, *XSetWMProperties()* calls *XSetWMIconName()*, which sets the WM_ICON_NAME property (see section 14.1.5). If the *argv* argument is non-NULL, *XSetWMProperties()* calls *XSetCommand()*, which sets the WM_COMMAND property (see section 14.2.1). Note that an *argc* of zero is allowed to indicate a zero-length command. Note also that the hostname of this machine is stored using *XSetWMClientMachine()* (see section 14.2.2).

If the *normal_hints* argument is non-NULL, *XSetWMProperties()* calls *XSetWMNormalHints()*, which sets the WM_NORMAL_HINTS property (see section 14.1.7). If the *wm_hints* argument is non-NULL, *XSetWMProperties()* calls *XSetWMHints()*, which sets the WM_HINTS property (see section 14.1.6).

If the *class_hints* argument is non-NULL, *XSetWMProperties()* calls *XSetClassHint()*, which sets the WM_CLASS property (see section 14.1.8). If the *res_name* member in the **XClassHint** structure is set to the NULL pointer and the *RESOURCE_NAME* environment variable is set, then the value of the environment variable is substituted for *res_name*. If the *res_name* member is NULL, the environment variable is not set, and *argv* and *argv[0]* are set, then the value of *argv[0]*, stripped of any directory prefixes, is substituted for *res_name*.

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

If the *window_name* argument is non-NULL, *XmbSetWMProperties()* sets the WM_NAME property. If the *icon_name* argument is non-NULL, *XmbSetWMProperties()* sets the WM_ICON_NAME property. The *window_name* and *icon_name* arguments are null-terminated strings in the encoding of the current locale. If the arguments can be fully converted to the STRING encoding, the properties are created with type “STRING”; otherwise, the arguments are converted to Compound Text, and the properties are created with type “COMPOUND_TEXT”.

If the *normal_hints* argument is non-NULL, *XmbSetWMProperties()* calls *XSetWMNormalHints()*, which sets the WM_NORMAL_HINTS property (see section 14.1.7). If the *wm_hints* argument is non-NULL, *XmbSetWMProperties()* calls *XSetWMHints()*, which sets the WM_HINTS property (see section 14.1.6).

If the *argv* argument is non-NULL, *XmbSetWMProperties()* sets the WM_COMMAND property from *argv* and *argc*. Note that an *argc* of 0 indicates a zero-length command.

The hostname of this machine is stored using *XSetWMClientMachine()* (see section 14.2.2).

If the *class_hints* argument is non-NULL, *XmbSetWMProperties()* sets the WM_CLASS property. If the *res_name* member in the **XClassHint** structure is set to the NULL pointer and the *RESOURCE_NAME* environment variable is set, the value of the environment variable is substituted for *res_name*. If the *res_name* member is NULL, the environment variable is not set, and *argv* and *argv[0]* are set, then the value of *argv[0]*, stripped of any directory prefixes, is substituted for *res_name*.

It is assumed that the supplied *class_hints.res_name* and *argv*, the *RESOURCE_NAME* environment variable, and the hostname of this machine are in the encoding of the locale announced for the *LC_CTYPE* category. (On POSIX-compliant systems, the *LC_CTYPE*, else *LANG* environment variable). The corresponding WM_CLASS, WM_COMMAND, and WM_CLIENT_MACHINE properties are typed according to the local host locale announcer. No encoding conversion is performed prior to storage in the properties.

For clients that need to process the property text in a locale, *XmbSetWMProperties()* sets the WM_LOCALE_NAME property to be the name of the current locale. The name is assumed to be in the Host Portable Character Encoding, and is converted to STRING for storage in the property.

XSetWMProperties() and *XmbSetWMProperties()* can generate BadAlloc and BadWindow errors.

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, null-separated, used to invoke the application.

WM_HINTS

Additional hints set by the client for use by the window manager. The C type of this property is **XWMHints**.

WM_ICON_NAME

The name to be used in an icon.

WM_NAME

The name of the application.

WM_NORMAL_HINTS

Size hints for a window in its normal state. The C type of this property is **XSizeHints**.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), *XAllocIconSize()*, *XAllocSizeHints()*, *XAllocWMHints()*, *XParseGeometry()*, *XSetCommand()*, *XSetTransientForHint()*, *XSetTextProperty()*, *XSetWMClientMachine()*, *XSetWMColormapWindows()*, *XSetWMIconName()*, *XSetWMName()*, *XSetWMProtocols()*, *XStringListToTextProperty()*, *XTextListToTextProperty()*

Xlib — C Language X Interface

NAME

XSetWMProtocols(), XGetWMProtocols() – set or read a window's WM_PROTOCOLS property

SYNOPSIS

```
Status XSetWMProtocols (Display *display, Window w,
                        Atom *protocols, int count)
Status XGetWMProtocols (Display *display, Window w,
                        Atom **protocols_return, int *count_return)
```

ARGUMENTS

display Specifies the connection to the X server.
count Specifies the number of protocols in the list.
count_return Returns the number of protocols in the list.
protocols Specifies the list of protocols.
protocols_return Returns the list of protocols.

DESCRIPTION

The *XSetWMProtocols()* function replaces the WM_PROTOCOLS property on the specified window with the list of atoms specified by the protocols argument. If the property does not already exist, *XSetWMProtocols()* sets the WM_PROTOCOLS property on the specified window to the list of atoms specified by the protocols argument. The property is stored with a type of **ATOM** and a format of 32. If it cannot intern the WM_PROTOCOLS atom, *XSetWMProtocols()* returns a zero status. Otherwise, it returns a nonzero status.

XSetWMProtocols() can generate BadAlloc and BadWindow errors.

The *XGetWMProtocols()* function returns the list of atoms stored in the WM_PROTOCOLS property on the specified window. These atoms describe window manager protocols in which the owner of this window is willing to participate. If the property exists, is of type **ATOM**, is of format 32, and the atom WM_PROTOCOLS can be interned, *XGetWMProtocols()* sets the *protocols_return* argument to a list of atoms, sets the *count_return* argument to the number of elements in the list, and returns a nonzero status. Otherwise, it sets neither of the return arguments and returns a zero status. To release the list of atoms, use *XFree()*.

XGetWMProtocols() can generate a BadWindow error.

PROPERTIES

WM_PROTOCOLS
 List of atoms that identify the communications protocols between the client and window manager in which the client is willing to participate.

DIAGNOSTICS

BadAlloc
 The server failed to allocate the requested resource or server memory.
 BadWindow
 A value for a Window argument does not name a defined Window.

SEE ALSO

XAllocClassHint(), XAllocIconSize(), XAllocSizeHints(), XAllocWMHints(), XFree(), XSetCommand(), XSetTransientForHint(), XSetTextProperty(), XSetWMClientMachine(), XSetWMColormapWindows(), XSetWMIconName(), XSetWMName(), XSetWMProperties(), XStringListToTextProperty()

Xlib — C Language X Interface

NAME

XStoreBytes(), XStoreBuffer(), XFetchBytes(), XFetchBuffer(), XRotateBuffers() – manipulate cut and paste buffers

SYNOPSIS

```
XStoreBytes (Display *display, char *bytes, int  nbytes)
XStoreBuffer (Display *display, char *bytes, int  nbytes,
              int  buffer)
char *XFetchBytes (Display *display, int *nbytes_return)
char *XFetchBuffer (Display *display, int *nbytes_return,
                   int  buffer)
XRotateBuffers (Display *display, int  rotate)
```

ARGUMENTS

buffer Specifies the buffer in which you want to store the bytes or from which you want the stored data returned.

bytes Specifies the bytes, which are not necessarily ASCII or null-terminated.

display Specifies the connection to the X server.

nbytes Specifies the number of bytes to be stored.

nbytes_return Returns the number of bytes in the buffer.

rotate Specifies how much to rotate the cut buffers.

DESCRIPTION

Note that the data can have embedded null characters, and need not be null terminated. The cut buffer's contents can be retrieved later by any client calling *XFetchBytes()*.

XStoreBytes() can generate a BadAlloc error.

If an invalid buffer is specified, the call has no effect. Note that the data can have embedded null characters, and need not be null terminated.

XStoreBuffer() can generate a BadAlloc error.

The *XFetchBytes()* function returns the number of bytes in the *nbytes_return* argument, if the buffer contains data. Otherwise, the function returns NULL and sets *nbytes* to 0. The appropriate amount of storage is allocated and the pointer returned. The client must free this storage when finished with it by calling *XFree()*.

The *XFetchBuffer()* function returns zero to the *nbytes_return* argument if there is no data in the buffer or if an invalid buffer is specified.

XFetchBuffer() can generate a BadValue error.

The *XRotateBuffers()* function rotates the cut buffers, such that buffer 0 becomes buffer *n*, buffer 1 becomes *n + 1 mod 8*, and so on. This cut buffer numbering is global to the display. Note that *XRotateBuffers()* generates BadMatch errors if any of the eight buffers have not been created.

XRotateBuffers() can generate a BadMatch error.

DIAGNOSTICS

BadAlloc

The server failed to allocate the requested resource or server memory.

BadAtom

A value for an Atom argument does not name a defined Atom.

BadMatch

Some argument or pair of arguments has the correct type and range but fails to match in some other way required by the request.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XFree()

Xlib — C Language X Interface

NAME

XStoreColors(), XStoreColor(), XStoreNamedColor() – set colors

SYNOPSIS

```
XStoreColors (Display *display, Colormap colormap,
              XColor color[], int ncolors)
XStoreColor (Display *display, Colormap colormap,
             XColor *color)
XStoreNamedColor (Display *display, Colormap colormap,
                  char * color, unsigned long pixel, int flags)
```

ARGUMENTS

color Specifies the pixel and RGB values or the color name string (for example, red).
color Specifies an array of color definition structures to be stored.
colormap Specifies the colormap.
display Specifies the connection to the X server.
flags Specifies which red, green, and blue components are set.
ncolors Specifies the number of **XColor** structures in the color definition array.
pixel Specifies the entry in the colormap.

DESCRIPTION

The *XStoreColors()* function changes the colormap entries of the pixel values specified in the pixel members of the **XColor** structures. You specify which color components are to be changed by setting DoRed, DoGreen, and / or DoBlue in the *flags* member of the **XColor** structures. If the *colormap* is an installed map for its screen, the changes are visible immediately. *XStoreColors()* changes the specified pixels if they are allocated writable in the colormap by any client, even if one or more pixels generates an error. If a specified pixel is not a valid index into the colormap, a BadValue error results. If a specified pixel either is unallocated or is allocated read-only, a BadAccess error results. If more than one pixel is in error, the one that gets reported is arbitrary.

XStoreColors() can generate BadAccess, BadColor, and BadValue errors.

The *XStoreColor()* function changes the colormap entry of the pixel value specified in the pixel member of the **XColor** structure. You specified this value in the *pixel* member of the **XColor** structure. This pixel value must be a read / write cell and a valid index into the colormap. If a specified pixel is not a valid index into the colormap, a BadValue error results. *XStoreColor()* also changes the red, green, and / or blue color components. You specify which color components are to be changed by setting DoRed, DoGreen, and / or DoBlue in the *flags* member of the **XColor** structure. If the colormap is an installed map for its screen, the changes are visible immediately.

XStoreColor() can generate BadAccess, BadColor, and BadValue errors.

The *XStoreNamedColor()* function looks up the named color with respect to the screen associated with the *colormap* and stores the result in the specified colormap. The *pixel* argument determines the entry in the colormap. The *flags* argument determines which of the red, green, and blue components are set. You can set this member to the bitwise inclusive OR of the bits DoRed, DoGreen, and DoBlue. If the color name is not in the Host Portable Character Encoding the result is implementation dependent. Use of uppercase or lowercase does not matter. If the specified *pixel* is not a valid index into the colormap, a BadValue error results. If the specified *pixel* either is unallocated or is

allocated read-only, a `BadAccess` error results.

`XStoreNamedColor()` can generate `BadAccess`, `BadColor`, `BadName`, and `BadValue` errors.

DIAGNOSTICS

`BadAccess`

A client attempted to free a color map entry that it did not already allocate.

`BadAccess`

A client attempted to store into a read-only color map entry.

`BadColor`

A value for a `Colormap` argument does not name a defined `Colormap`.

`BadName`

A font or color of the specified name does not exist.

`BadValue`

Some numeric value falls outside the range of values accepted by the request.

Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

`XAllocColor()`, `XCreateColormap()`, `XQueryColor()`

Xlib — C Language X Interface

NAME

XStringListToTextProperty(), XTextPropertyToStringList(), XFreeStringList(),
XTextProperty() – convert string lists and text property structure

SYNOPSIS

```
Status XStringListToTextProperty (char **list, int count,
                                   XTextProperty *text_prop_return)
Status XTextPropertyToStringList (XTextProperty *text_prop,
                                   char ***list_return,
                                   int *count_return)
void XFreeStringList (char **list)
```

ARGUMENTS

count Specifies the number of strings.
count_return Returns the number of strings.
list Specifies the list of strings to be freed.
list Specifies a list of null-terminated character strings.
list_return Returns a list of null-terminated character strings.
text_prop Specifies the **XTextProperty** structure to be used.
text_prop_return Returns the **XTextProperty** structure.

DESCRIPTION

The *XStringListToTextProperty()* function sets the specified **XTextProperty** to be of type STRING (format 8) with a value representing the concatenation of the specified list of null-separated character strings. An extra null byte (which is not included in the *nitems* member) is stored at the end of the value field of *text_prop_return*. The strings are assumed (without verification) to be in the STRING encoding. If insufficient memory is available for the new value string, *XStringListToTextProperty()* does not set any fields in the **XTextProperty** structure and returns a zero status. Otherwise, it returns a nonzero status. To free the storage for the value field, use **XFree**.

The *XTextPropertyToStringList()* function returns a list of strings representing the null-separated elements of the specified **XTextProperty** structure. The data in *text_prop* must be of type STRING and format 8. Multiple elements of the property (for example, the strings in a disjoint text selection) are separated by NULL (encoding 0). The contents of the property are not null-terminated. If insufficient memory is available for the list and its elements, *XTextPropertyToStringList()* sets no return values and returns a zero status. Otherwise, it returns a nonzero status. To free the storage for the list and its contents, use *XFreeStringList()*.

The *XFreeStringList()* function releases memory allocated by *XmbTextPropertyToTextList()* and *XTextPropertyToStringList()*, and the missing charset list allocated by *XCreateFontSet()*.

STRUCTURES

The **XTextProperty** structure contains:

```
typedef struct {
    unsigned char *value;    property data
```

```
        Atom encoding;           type of property
        int format;              8, 16, or 32
        unsigned long nitems;    number of items in value
    } XTextProperty;
```

SEE ALSO

XAllocClassHint(), XAllocIconSize(), XAllocSizeHints(), XAllocWMHints(), XFree(), XSetCommand(), XSetTransientForHint(), XSetTextProperty(), XSetWMClientMachine(), XSetWMColormapWindows(), XSetWMIconName(), XSetWMName(), XSetWMProperties(), XSetWMProtocols()

Xlib — C Language X Interface

NAME

XStringToKeysym(), XKeysymToString(), XKeycodeToKeysym(), XKeysymToKeycode()
– convert keysyms

SYNOPSIS

```
KeySym XStringToKeysym (char *string)
char *XKeysymToString (KeySym keysym)
KeySym XKeycodeToKeysym (Display *display, KeyCode keycode,
                        int index)
KeyCode XKeysymToKeycode (Display *display, KeySym keysym)
```

ARGUMENTS

display Specifies the connection to the X server.
index Specifies the element of KeyCode vector.
keycode Specifies the KeyCode.
keysym Specifies the KeySym that is to be searched for or converted.
string Specifies the name of the KeySym that is to be converted.

DESCRIPTION

Standard KeySym names are obtained from <X11/keysymdef.h> by removing the *XK_* prefix from each name. KeySyms that are not part of the Xlib standard also may be obtained with this function. Note that the set of KeySyms that are available in this manner and the mechanisms by which Xlib obtains them is implementation dependent.

If the keysym name is not in the Host Portable Character Encoding the result is implementation dependent. If the specified string does not match a valid KeySym, *XStringToKeysym()* returns NoSymbol.

The returned string is in a static area and must not be modified. The returned string is in the Host Portable Character Encoding. If the specified KeySym is not defined, *XKeysymToString()* returns NULL.

The *XKeycodeToKeysym()* function uses internal Xlib tables and returns the KeySym defined for the specified KeyCode and the element of the KeyCode vector. If no symbol is defined, *XKeycodeToKeysym()* returns NoSymbol.

If the specified KeySym is not defined for any KeyCode, *XKeysymToKeycode()* returns zero.

SEE ALSO

XLookupKeysym()

Xlib — C Language X Interface

NAME

XSupportsLocale(), XSetLocaleModifiers() – determine locale support and configure locale modifiers

SYNOPSIS

```
Bool XSupportsLocale (void)
char *XSetLocaleModifiers (char *modifier_list)
```

ARGUMENTS

modifier_list
Specifies the modifiers.

DESCRIPTION

The *XSupportsLocale()* function returns True if Xlib functions are capable of operating under the current locale. If it returns False, Xlib locale-dependent functions for which the *XLocaleNotSupported* return status is defined will return *XLocaleNotSupported*. Other Xlib locale-dependent routines will operate in the “C” locale.

XSetLocaleModifiers() sets the X modifiers for the current locale setting. The *modifier_list* argument is a null-terminated string of the form

```
{@ category = value }
```

That is, it has zero or more concatenated “@ *category* = *value* ” entries where *category* is a category name and *value* is the (possibly empty) setting for that category. The values are encoded in the current locale. Category names are restricted to the POSIX Portable Filename Character Set.

The local host X locale modifiers announcer (on POSIX-compliant systems, the *XMODIFIERS* environment variable) is appended to the *modifier_list* to provide default values on the local host. If a given category appears more than once in the list, the first setting in the list is used. If a given category is not included in the full modifier list, the category is set to an implementation-dependent default for the current locale. An empty value for a category explicitly specifies the implementation-dependent default.

If the function is successful, it returns a pointer to a string. The contents of the string are such that a subsequent call with that string (in the same locale) will restore the modifiers to the same settings. If *modifier_list* is a NULL pointer, *XSetLocaleModifiers()* also returns a pointer to such a string, and the current locale modifiers are not changed.

If invalid values are given for one or more modifier categories supported by the locale, a NULL pointer is returned, and none of the current modifiers are changed.

At program startup the modifiers that are in effect are unspecified until the first successful call to set them. Whenever the locale is changed, the modifiers that are in effect become unspecified until the next successful call to set them. Clients should always call *XSetLocaleModifiers()* with a non-NULL *modifier_list* after setting the locale, before they call any locale-dependent Xlib routine.

The only standard modifier category currently defined is “im”, which identifies the desired input method. The values for input method are not standardized. A single locale may use multiple input methods, switching input method under user control. The modifier may specify the initial input method in effect, or an ordered list of input methods. Multiple input methods may be specified in a single im value string in an implementation-dependent manner.

The returned modifiers string is owned by Xlib and should not be modified or freed by the client. It may be freed by Xlib after the current locale or modifiers is changed. Until freed, it will not be modified by Xlib.

SEE ALSO

Xlib — C Language X Interface

NAME

XSynchronize(), XSetAfterFunction() – enable or disable synchronization

SYNOPSIS

```
int (Display *display, Bool onoff)  
int (Display *display, int (* procedure)())
```

ARGUMENTS

display Specifies the connection to the X server.

procedure

Specifies the function to be called.

onoff Specifies a Boolean value that indicates whether to enable or disable synchronization.

DESCRIPTION

The *XSynchronize()* function returns the previous after function. If *onoff* is True, *XSynchronize()* turns on synchronous behavior. If *onoff* is False, *XSynchronize()* turns off synchronous behavior.

The specified procedure is called with only a display pointer. *XSetAfterFunction()* returns the previous after function.

SEE ALSO

XSetErrorHandler()

Xlib — C Language X Interface

NAME

XTextExtents(), XTextExtents16(), XQueryTextExtents(), XQueryTextExtents16() – compute or query text extents

SYNOPSIS

```
XTextExtents (XFontStruct *font_struct, char *string, int nchars,
              int *direction_return, int *font_ascent_return,
              int *font_descent_return, XCharStruct *overall_return)
XTextExtents16 (XFontStruct *font_struct, XChar2b *string,
                int nchars, int *direction_return,
                int *font_ascent_return, int *font_descent_return,
                XCharStruct *overall_return)
XQueryTextExtents (Display *display, XID font_ID, char *string,
                  int nchars, int *direction_return,
                  int *font_ascent_return, int *font_descent_return,
                  XCharStruct *overall_return)
XQueryTextExtents16 (Display *display, XID font_ID,
                    XChar2b *string, int nchars,
                    int *direction_return, int *font_ascent_return,
                    int *font_descent_return,
                    XCharStruct *overall_return)
```

ARGUMENTS

direction_return
Returns the value of the direction hint (FontLeftToRight or FontRightToLeft).

display
Specifies the connection to the X server.

font_ID
Specifies either the font ID or the GContext ID that contains the font.

font_ascent_return
Returns the font ascent.

font_descent_return
Returns the font descent.

font_struct
Specifies the **XFontStruct** structure.

nchars
Specifies the number of characters in the character string.

string
Specifies the character string.

overall_return
Returns the overall size in the specified **XCharStruct** structure.

DESCRIPTION

The *XTextExtents()* and *XTextExtents16()* functions perform the size computation locally and, thereby, avoid the round-trip overhead of *XQueryTextExtents()* and *XQueryTextExtents16()*. Both functions return an **XCharStruct** structure, whose members are set to the values as follows.

The *ascent* member is set to the maximum of the ascent metrics of all characters in the string. The *descent* member is set to the maximum of the descent metrics. The *width* member is set to the sum of the character-width metrics of all characters in the string. For each character in the string, let *W* be the sum of the character-width metrics of all characters preceding it in the string. Let *L* be the left-side-bearing metric of the character plus *W*. Let *R* be the right-side-bearing metric of the character plus *W*. The *lbearing* member is set to the minimum *L* of all characters in the string. The *rbearing* member is

set to the maximum R.

For fonts defined with linear indexing rather than 2-byte matrix indexing, each **XChar2b** structure is interpreted as a 16-bit number with *byte1* as the most-significant byte. If the font has no defined default character, undefined characters in the string are taken to have all zero metrics.

The *XQueryTextExtents()* and *XQueryTextExtents16()* functions return the bounding box of the specified 8-bit and 16-bit character string in the specified font or the font contained in the specified GC. These functions query the X server and, therefore, suffer the round-trip overhead that is avoided by *XTextExtents()* and *XTextExtents16()*. Both functions return a **XCharStruct** structure, whose members are set to the values as follows.

The *ascent* member is set to the maximum of the ascent metrics of all characters in the string. The *descent* member is set to the maximum of the descent metrics. The *width* member is set to the sum of the character-width metrics of all characters in the string. For each character in the string, let W be the sum of the character-width metrics of all characters preceding it in the string. Let L be the left-side-bearing metric of the character plus W. Let R be the right-side-bearing metric of the character plus W. The *lbearing* member is set to the minimum L of all characters in the string. The *rbearing* member is set to the maximum R.

For fonts defined with linear indexing rather than 2-byte matrix indexing, each **XChar2b** structure is interpreted as a 16-bit number with *byte1* as the most-significant byte. If the font has no defined default character, undefined characters in the string are taken to have all zero metrics.

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

XQueryTextExtents() and *XQueryTextExtents16()* can generate BadFont and BadGC errors.

DIAGNOSTICS

BadFont

A value for a Font or GContext argument does not name a defined Font.

BadGC A value for a GContext argument does not name a defined GContext.

SEE ALSO

XLoadFont(), *XTextWidth()*

Xlib — C Language X Interface

NAME

XTextWidth(), XTextWidth16() – compute text width

SYNOPSIS

```
int XTextWidth (XFontStruct *font_struct, char *string, int count)
int XTextWidth16 (XFontStruct *font_struct, XChar2b *string,
                  int count)
```

ARGUMENTS

count Specifies the character count in the specified string.
font_struct
 Specifies the font used for the width computation.
string Specifies the character string.

DESCRIPTION

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

SEE ALSO

XLoadFont(), *XTextExtents()*
Xlib — C Language X Interface

NAME

XTranslateCoordinates() – translate window coordinates

SYNOPSIS

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

ARGUMENTS

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

dest_w Specifies the destination window.

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

display Specifies the connection to the X server.

src_w Specifies the source window.

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

DESCRIPTION

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

XTranslateCoordinates() can generate a BadWindow error.

DIAGNOSTICS

BadWindow
A value for a Window argument does not name a defined Window.

SEE ALSO

Xlib — C Language X Interface

NAME

XUnmapEvent – UnmapNotify event structure

STRUCTURES

The structure for UnmapNotify events contains:

```
typedef struct {
    int type;                UnmapNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;        true if this came from a SendEvent request
    Display *display;       Display the event was read from
    Window event;
    Window window;
    Bool from_configure;
} XUnmapEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *event* member is set either to the unmapped window or to its parent, depending on whether StructureNotify or SubstructureNotify was selected. This is the window used by the X server to report the event. The *window* member is set to the window that was unmapped. The *from_configure* member is set to True if the event was generated as a result of a resizing of the window's parent when the window itself had a *win_gravity* of UnmapGravity.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*,
XCirculateRequestEvent(), *XColormapEvent()*, *XConfigureEvent()*,
XConfigureRequestEvent(), *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*,
XExposeEvent(), *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*,
XKeymapEvent(), *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*,
XReparentEvent(), *XResizeRequestEvent()*, *XSelectionClearEvent()*, *XSelectionEvent()*,
XSelectionRequestEvent(), *XVisibilityEvent()*

Xlib — C Language X Interface

NAME

XUnmapWindow(), XUnmapSubwindows() – unmap windows

SYNOPSIS

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

ARGUMENTS

display Specifies the connection to the X server.
w Specifies the window.

DESCRIPTION

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

XUnmapWindow() can generate a BadWindow error.

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

XUnmapSubwindows() can generate a BadWindow error.

DIAGNOSTICS

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XChangeWindowAttributes(), *XConfigureWindow()*, *XCreateWindow()*,
XDestroyWindow(), *XMapWindow()* *XRaiseWindow()*

Xlib — C Language X Interface

NAME

XVaCreateNestedList() – allocate a nested variable argument list

SYNOPSIS

```
typedef void * XVaNestedList  
XVaNestedList XVaCreateNestedList (int dummy, ... )
```

ARGUMENTS

dummy Unused argument (required by ANSI C).
... Specifies the variable length argument list.

DESCRIPTION

The *XVaCreateNestedList()* function allocates memory and copies its arguments into a single list pointer which may be used as value for arguments requiring a list value. Any entries are copied as specified. Data passed by reference is not copied; the caller must ensure data remains valid for the lifetime of the nested list. The list should be freed using *XFree()* when it is no longer needed.

SEE ALSO

Xlib — C Language X Interface

NAME

XVisibilityNotifyEvent – VisibilityNotify event structure

STRUCTURES

The structure for VisibilityNotify events contains:

```
typedef struct {
    int type;                VisibiltyNotify
    unsigned long serial;    # of last request processed by server
    Bool send_event;         true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Window window;
    int state;
} XVisibilityEvent;
```

When you receive this event, the structure members are set as follows.

The *type* member is set to the event type constant name that uniquely identifies it. For example, when the X server reports a GraphicsExpose event to a client application, it sends an **XGraphicsExposeEvent** structure with the *type* member set to GraphicsExpose. The *display* member is set to a pointer to the display the event was read on. The *send_event* member is set to True if the event came from a SendEvent protocol request. The *serial* member is set from the serial number reported in the protocol but expanded from the 16-bit least-significant bits to a full 32-bit value. The *window* member is set to the window that is most useful to toolkit dispatchers.

The *window* member is set to the window whose visibility state changes. The *state* member is set to the state of the window's visibility and can be VisibilityUnobscured, VisibilityPartiallyObscured, or VisibilityFullyObscured. The X server ignores all of a window's subwindows when determining the visibility state of the window and processes VisibilityNotify events according to the following:

- When the window changes state from partially obscured, fully obscured, or not viewable to viewable and completely unobscured, the X server generates the event with the *state* member of the **XVisibilityEvent** structure set to VisibilityUnobscured.
- When the window changes state from viewable and completely unobscured or not viewable to viewable and partially obscured, the X server generates the event with the *state* member of the **XVisibilityEvent** structure set to VisibilityPartiallyObscured.
- When the window changes state from viewable and completely unobscured, viewable and partially obscured, or not viewable to viewable and fully obscured, the X server generates the event with the *state* member of the **XVisibilityEvent** structure set to VisibilityFullyObscured.

SEE ALSO

XAnyEvent(), *XButtonEvent()*, *XCreateWindowEvent()*, *XCirculateEvent()*, *XCirculateRequestEvent()*, *XColormapEvent()*, *XConfigureEvent()*, *XConfigureRequestEvent()*, *XCrossingEvent()*, *XDestroyWindowEvent()*, *XErrorEvent()*, *XExposeEvent()*, *XFocusChangeEvent()*, *XGraphicsExposeEvent()*, *XGravityEvent()*, *XKeymapEvent()*, *XMapEvent()*, *XMapRequestEvent()*, *XPropertyEvent()*, *XReparentEvent()*, *XResizeRequestEvent()*, *XSelectionClearEvent()*, *XSelectionEvent()*, *XSelectionRequestEvent()*, *XUnmapEvent()*,

NAME

XWarpPointer() – move pointer

SYNOPSIS

```
XWarpPointer (Display *display, Window src_w, Window dest_w,
             int src_x, int src_y, unsigned int src_width,
             unsigned int src_height, int dest_x, int dest_y)
```

ARGUMENTS

dest_w Specifies the destination window or None.
dest_x
dest_y Specify the x and y coordinates within the destination window.
display Specifies the connection to the X server.
src_x
src_y
src_width
src_height Specify a rectangle in the source window.
src_w Specifies the source window or None.

DESCRIPTION

If *dest_w* is None, *XWarpPointer()* moves the pointer by the offsets (*dest_x*, *dest_y*) relative to the current position of the pointer. If *dest_w* is a window, *XWarpPointer()* moves the pointer to the offsets (*dest_x*, *dest_y*) relative to the origin of *dest_w*. However, if *src_w* is a window, the move only takes place if the window *src_w* contains the pointer and if the specified rectangle of *src_w* contains the pointer.

The *src_x* and *src_y* coordinates are relative to the origin of *src_w*. If *src_height* is zero, it is replaced with the current height of *src_w* minus *src_y*. If *src_width* is zero, it is replaced with the current width of *src_w* minus *src_x*.

There is seldom any reason for calling this function. The pointer should normally be left to the user. If you do use this function, however, it generates events just as if the user had instantaneously moved the pointer from one position to another. Note that you cannot use *XWarpPointer()* to move the pointer outside the *confine_to* window of an active pointer grab. An attempt to do so will only move the pointer as far as the closest edge of the *confine_to* window.

XWarpPointer() can generate a BadWindow error.

DIAGNOSTICS

BadWindow
 A value for a Window argument does not name a defined Window.

SEE ALSO

XSetInputFocus()
Xlib — C Language X Interface

NAME

XcmsAllocColor(), XcmsAllocNamedColor() – allocate colors

SYNOPSIS

```
Status XcmsAllocColor (Display *display, Colormap colormap,
                      XcmsColor *color_in_out,
                      XcmsColorFormat result_format)
Status XcmsAllocNamedColor (Display *display, Colormap colormap,
                           char *color_string,
                           XcmsColorFormat result_format,
                           XcmsColor *color_screen_return,
                           XcmsColor *color_exact_return)
```

ARGUMENTS

display Specifies the connection to the X server.

colormap Specifies the colormap.

color_exact_return Returns the color specification parsed from the color string or parsed from the corresponding string found in a color name database.

color_in_out Specifies the color to allocate and returns the pixel and color that is actually used in the colormap.

color_screen_return Returns the pixel value of the color cell and color specification that actually is stored for that cell.

color_string Specifies the color string whose color definition structure is to be returned.

result_format Specifies the color format for the returned color specification.

DESCRIPTION

The *XcmsAllocColor()* function is similar to *XAllocColor()* except the color can be specified in any format. The *XcmsAllocColor()* function ultimately calls *XAllocColor()* to allocate a read-only color cell (colormap entry) with the specified color. *XcmsAllocColor()* first converts the color specified to an RGB value and then passes this to *XAllocColor()*. *XcmsAllocColor()* returns the pixel value of the color cell and the color specification actually allocated. This returned color specification is the result of converting the RGB value returned by *XAllocColor()* into the format specified with the *result_format* argument. If there is no interest in a returned color specification, unnecessary computation can be bypassed if *result_format* is set to *XcmsRGBFormat*. The corresponding colormap cell is read-only. If this routine returns *XcmsFailure()*, the *color_in_out* color specification is left unchanged.

XcmsAllocColor() can generate a BadColor errors.

The *XcmsAllocNamedColor()* function is similar to *XAllocNamedColor()* except the color returned can be in any format specified. This function ultimately calls *XAllocColor()* to allocate a read-only color cell with the color specified by a color string. The color string is parsed into an **XcmsColor** structure (see *XcmsLookupColor()*), converted to an RGB value, then finally passed to the *XAllocColor()*. If the color name is not in the Host Portable Character Encoding the result is implementation dependent. Use of uppercase

or lowercase does not matter.

This function returns both the color specification as a result of parsing (exact specification) and the actual color specification stored (screen specification). This screen specification is the result of converting the RGB value returned by *XAllocColor()* into the format specified in *result_format*. If there is no interest in a returned color specification, unnecessary computation can be bypassed if *result_format* is set to *XcmsRGBFormat*.

XcmsAllocNamedColor() can generate a BadColor errors.

DIAGNOSTICS

BadColor

A value for a Colormap argument does not name a defined Colormap.

SEE ALSO

XcmsQueryColor(), *XcmsStoreColor()*

Xlib — C Language X Interface

NAME

XcmsCCOfColormap(), XcmsSetCCOfColormap() – query and modify CCC of a colormap

SYNOPSIS

```
XcmsCCC XcmsCCOfColormap (Display *display, Colormap colormap)
XcmsCCC XcmsSetCCOfColormap (Display *display, Colormap colormap,
                             XcmsCCC ccc)
```

ARGUMENTS

display Specifies the connection to the X server.
ccc Specifies the CCC.
colormap Specifies the colormap.

DESCRIPTION

The *XcmsCCOfColormap()* function returns the CCC associated with the specified colormap. Once obtained, the CCC attributes can be queried or modified. Unless the CCC associated with the specified colormap is changed with *XcmsSetCCOfColormap()*, this CCC is used when the specified colormap is used as an argument to color functions.

The *XcmsSetCCOfColormap()* function changes the CCC associated with the specified colormap. It returns the CCC previously associated to the colormap. If they are not used again in the application, CCCs should be freed by calling *XcmsFreeCCC()*.

SEE ALSO

DisplayOfCCC(), *XcmsConvertColors()*, *XcmsCreateCCC()*, *XcmsDefaultCCC()*, *XcmsSetWhitePoint()*

Xlib — C Language X Interface

NAME

XcmsCIELabQueryMaxC(), XcmsCIELabQueryMaxL(), XcmsCIELabQueryMaxLC(), XcmsCIELabQueryMinL() – obtain the CIE L*a*b* coordinates

SYNOPSIS

```
Status XcmsCIELabQueryMaxC (XcmsCCC ccc, XcmsFloat hue_angle,
                             XcmsFloat L_star,
                             XcmsColor *color_return)
Status XcmsCIELabQueryMaxL (XcmsCCC ccc, XcmsFloat hue_angle,
                             XcmsFloat chroma,
                             XcmsColor *color_return)
Status XcmsCIELabQueryMaxLC (XcmsCCC ccc, XcmsFloat hue_angle,
                              XcmsColor *color_return)
Status XcmsCIELabQueryMinL (XcmsCCC ccc, XcmsFloat hue_angle,
                              XcmsFloat chroma,
                              XcmsColor *color_return)
```

ARGUMENTS

ccc Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

chroma Specifies the chroma at which to find maximum lightness (*MaxL*) or minimum lightness (*MinL*).

color_return Returns the CIE L*a*b* coordinates of maximum chroma (*MaxC* and *MaxLC*), maximum lightness (*MaxL*), or minimum lightness (*MinL*) displayable by the screen for the given hue angle and lightness (*MaxC*), hue angle and chroma (*MaxL* and *MinL*), or hue angle (*MaxLC*). The white point associated with the returned color specification is the Screen White Point. The value returned in the *pixel* member is undefined.

hue_angle Specifies the hue angle in degrees at which to find maximum chroma (*MaxC* and *MaxLC*), maximum lightness (*MaxL*), or minimum lightness (*MinL*).

L_star Specifies the lightness (*L**) at which to find maximum chroma (*MaxC*).

DESCRIPTION

The *XcmsCIELabQueryMaxC()* function, given a hue angle and lightness, finds the point of maximum chroma displayable by the screen. It returns this point in CIE L*a*b* coordinates.

The *XcmsCIELabQueryMaxL()* function, given a hue angle and chroma, finds the point in CIE L*a*b* color space of maximum lightness (*L**) displayable by the screen. It returns this point in CIE L*a*b* coordinates. An *XcmsFailure* return value usually indicates that the given chroma is beyond maximum for the given hue angle.

The *XcmsCIELabQueryMaxLC()* function, given a hue angle, finds the point of maximum chroma displayable by the screen. It returns this point in CIE L*a*b* coordinates.

The *XcmsCIELabQueryMinL()* function, given a hue angle and chroma, finds the point of minimum lightness (*L**) displayable by the screen. It returns this point in CIE L*a*b* coordinates. An *XcmsFailure()* return value usually indicates that the given chroma is beyond maximum for the given hue angle.

SEE ALSO

XcmsCIELuvQueryMaxC(), *XcmsTekHVCQueryMaxC()*, *XcmsQueryBlack()*

Xlib — C Language X Interface

NAME

XcmsCIELuvQueryMaxC(), XcmsCIELuvQueryMaxL(), XcmsCIELuvQueryMaxLC(), XcmsCIELuvQueryMinL() – obtain the CIE L*u*v* coordinates

SYNOPSIS

```
Status XcmsCIELuvQueryMaxC (XcmsCCC ccc, XcmsFloat hue_angle,
                             XcmsFloat L_star,
                             XcmsColor *color_return)
Status XcmsCIELuvQueryMaxL (XcmsCCC ccc, XcmsFloat hue_angle,
                             XcmsFloat chroma,
                             XcmsColor *color_return)
Status XcmsCIELuvQueryMaxLC (XcmsCCC ccc, XcmsFloat hue_angle,
                              XcmsColor *color_return)
Status XcmsCIELuvQueryMinL (XcmsCCC ccc, XcmsFloat hue_angle,
                             XcmsFloat chroma,
                             XcmsColor *color_return)
```

ARGUMENTS

ccc Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

chroma Specifies the chroma at which to find maximum lightness (*MaxL*) or minimum lightness (*MinL*).

color_return Returns the CIE L*u*v* coordinates of maximum chroma (*MaxC* and *MaxLC*), maximum lightness (*MaxL*), or minimum lightness (*MinL*) displayable by the screen for the given hue angle and lightness (*MaxC*), hue angle and chroma (*MaxL* and *MinL*), or hue angle (*MaxLC*). The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

hue_angle Specifies the hue angle in degrees at which to find maximum chroma (*MaxC* and *MaxLC*), maximum lightness (*MaxL*), or minimum lightness (*MinL*).

L_star Specifies the lightness (*L**) at which to find maximum chroma (*MaxC*) or maximum lightness (*MaxL*).

DESCRIPTION

The *XcmsCIELuvQueryMaxC()* function, given a hue angle and lightness, finds the point of maximum chroma displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates.

The *XcmsCIELuvQueryMaxL()* function, given a hue angle and chroma, finds the point in CIE L*u*v* color space of maximum lightness (*L**) displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates. An *XcmsFailure* return value usually indicates that the given chroma is beyond maximum for the given hue angle.

The *XcmsCIELuvQueryMaxLC()* function, given a hue angle, finds the point of maximum chroma displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates.

The *XcmsCIELuvQueryMinL()* function, given a hue angle and chroma, finds the point of minimum lightness (*L**) displayable by the screen. Note that it returns this point in CIE L*u*v* coordinates. An *XcmsFailure()* return value usually indicates that the given

chroma is beyond maximum for the given hue angle.

SEE ALSO

XcmsCIELabQueryMaxC(), *XcmsTekHVCQueryMaxC()*, *XcmsQueryBlack()*

Xlib — C Language X Interface

NAME

XcmsColor, XcmsRGB, XcmsRGBi, XcmsCIEXYZ, XcmsCIEuvY, XcmsCIExyY, XcmsCIELab, XcmsCIELuv, XcmsTekHVC, XcmsPad – Xcms color structure

STRUCTURES

The structure for **XcmsColor** contains:

typedef unsigned long XcmsColorFormat; Color Specification Format

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

typedef double XcmsFloat;

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

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

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

```
typedef struct {
    XcmsFloat u_prime;        0.0 to ~0.6
    XcmsFloat v_prime;        0.0 to ~0.6
    XcmsFloat Y;              0.0 to 1.0
} XcmsCIEuvY;                   CIE u'v'Y
```

```
typedef struct {
    XcmsFloat x;              0.0 to ~.75
    XcmsFloat y;              0.0 to ~.85
```

```

        XcmsFloat Y;           0.0 to 1.0
    } XcmsCIExyY;              CIE xyY

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

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

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

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

```

DESCRIPTION

The **XcmsColor** structure contains a union of substructures, each supporting color specification encoding for a particular color space.

SEE ALSO

XcmsAllocColor(), XcmsStoreColor(), XcmsConvertColors()

Xlib — C Language X Interface

NAME

XcmsConvertColors() – convert CCC color specifications

SYNOPSIS

```
Status XcmsConvertColors (XcmsCCC ccc, XcmsColor colors_in_out[],
                          unsigned int ncolors,
                          XcmsColorFormat target_format,
                          Bool compression_flags_return[])
```

ARGUMENTS

- ccc* Specifies the CCC. If Conversion is between device-independent color spaces only (for example, TekHVC to CIE Luv), the CCC is necessary only to specify the Client White Point.
- colors_in_out* Specifies an array of color specifications. Pixel members are ignored and remain unchanged upon return.
- compression_flags_return* Specifies an array of Boolean values for returning compression status. If a non-NULL pointer is supplied, each element of the array is set to True if the corresponding color was compressed, and False otherwise. Pass NULL if the compression status is not useful.
- ncolors* Specifies the number of **XcmsColor** structures in the color specification array.
- target_format* Specifies the target color specification format.

DESCRIPTION

The *XcmsConvertColors()* function converts the color specifications in the specified array of **XcmsColor** structures from their current format to a single target format, using the specified CCC. When the return value is *XcmsFailure*, the contents of the color specification array are left unchanged.

The array may contain a mixture of color specification formats (for example, 3 CIE XYZ, 2 CIE Luv, ...). Note that when the array contains both device-independent and device-dependent color specifications, and the *target_format* argument specifies a device-dependent format (for example, *XcmsRGBiFormat*, *XcmsRGBFormat*) all specifications are converted to CIE XYZ format then to the target device-dependent format.

SEE ALSO

DisplayOfCCC(), *XcmsCCCOFColormap()*, *XcmsCreateCCC()*, *XcmsDefaultCCC()*, *XcmsSetWhitePoint()*

Xlib — C Language X Interface

NAME

XcmsCreateCCC(), XcmsFreeCCC() – creating and destroying CCCs

SYNOPSIS

```
XcmsCCC XcmsCreateCCC (Display *display, int screen_number,
                      Visual *visual,
                      XcmsColor *client_white_point,
                      XcmsCompressionProc compression_proc,
                      XPointer compression_client_data,
                      XcmsWhiteAdjustProc white_adjust_proc,
                      XPointer white_adjust_client_data)

void XcmsFreeCCC (XcmsCCC ccc)
```

ARGUMENTS

display Specifies the connection to the X server.

ccc Specifies the CCC.

client_white_point
Specifies the Client White Point. If NULL, the Client White Point is to be assumed to be the same as the Screen White Point. Note that the *pixel* member is ignored.

compression_client_data
Specifies client data for use by the gamut compression procedure or NULL.

compression_proc
Specifies the gamut compression procedure that is to be applied when a color lies outside the screen's color gamut. If NULL and when functions using this CCC must convert a color specification to a device-dependent format and encounters a color that lies outside the screen's color gamut, that function will return XcmsFailure.

screen_number
Specifies the appropriate screen number on the host server.

visual Specifies the visual type.

white_adjust_client_data
Specifies client data for use with the white point adjustment procedure or NULL.

white_adjust_proc
Specifies the white adjustment procedure that is to be applied when the Client White Point differs from the Screen White Point. NULL indicates that no white point adjustment is desired.

DESCRIPTION

The *XcmsCreateCCC()* function creates a CCC for the specified display, screen, and visual.

The *XcmsFreeCCC()* function frees the memory used for the specified CCC. Note that default CCCs and those currently associated with colormaps are ignored.

SEE ALSO

DisplayOfCCC(), *XcmsCCCOFColormap()*, *XcmsConvertColors()*, *XcmsDefaultCCC()*, *XcmsSetWhitePoint()*

Xlib — C Language X Interface

NAME

XcmsDefaultCCC() – obtain the default CCC for a screen

SYNOPSIS

XcmsCCC XcmsDefaultCCC (Display **display*, int *screen_number*)

ARGUMENTS

display Specifies the connection to the X server.

screen_number

Specifies the appropriate screen number on the host server.

DESCRIPTION

The *XcmsDefaultCCC()* function returns the default CCC for the specified screen. Its visual is the default visual of the screen. Its initial gamut compression and white point adjustment procedures as well as the associated client data are implementation specific.

SEE ALSO

DisplayOfCCC(), *XcmsCCCOFColormap()*, *XcmsConvertColors()*, *XcmsCreateCCC()*, *XcmsSetWhitePoint()*

Xlib — C Language X Interface

NAME

XcmsQueryBlack(), *XcmsQueryBlue()*, *XcmsQueryGreen()*, *XcmsQueryRed()*, *XcmsQueryWhite()* – obtain black, blue, green, red, and white CCC color specifications

SYNOPSIS

```
Status XcmsQueryBlack (XcmsCCC ccc, XcmsColorFormat target_format,
                      XcmsColor *color_return)
Status XcmsQueryBlue (XcmsCCC ccc, XcmsColorFormat target_format,
                     XcmsColor *color_return)
Status XcmsQueryGreen (XcmsCCC ccc, XcmsColorFormat target_format,
                      XcmsColor *color_return)
Status XcmsQueryRed (XcmsCCC ccc, XcmsColorFormat target_format,
                    XcmsColor *color_return)
Status XcmsQueryWhite (XcmsCCC ccc, XcmsColorFormat target_format,
                      XcmsColor *color_return)
```

ARGUMENTS

ccc Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

color_return Returns the color specification in the specified target format. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

target_format Specifies the target color specification format.

DESCRIPTION

The *XcmsQueryBlack()* function returns the color specification in the specified target format for zero intensity red, green, and blue.

The *XcmsQueryBlue()* function returns the color specification in the specified target format for full intensity blue while red and green are zero.

The *XcmsQueryGreen()* function returns the color specification in the specified target format for full intensity green while red and blue are zero.

The *XcmsQueryRed()* function returns the color specification in the specified target format for full intensity red while green and blue are zero.

The *XcmsQueryWhite()* function returns the color specification in the specified target format for full intensity red, green, and blue.

SEE ALSO

XcmsCIELabQueryMaxC(), *XcmsCIELuvQueryMaxC()*, *XcmsTekHVCQueryMaxC()*

Xlib — C Language X Interface

NAME

XcmsQueryColor(), XcmsQueryColors(), XcmsLookupColor() – obtain color values

SYNOPSIS

```
Status XcmsQueryColor (Display *display, Colormap colormap,
                        XcmsColor *color_in_out,
                        XcmsColorFormat result_format)
Status XcmsQueryColors (Display *display, Colormap colormap,
                        XcmsColor colors_in_out[],
                        unsigned int ncolors,
                        XcmsColorFormat result_format)
Status XcmsLookupColor (Display *display, Colormap colormap,
                        char *color_string,
                        XcmsColor *color_exact_return,
                        XcmsColor *color_screen_return,
                        XcmsColorFormat result_format)
```

ARGUMENTS

display Specifies the connection to the X server.

colormap Specifies the colormap.

color_exact_return Returns the color specification parsed from the color string or parsed from the corresponding string found in a color name database.

color_in_out Specifies the pixel member that indicates the color cell to query, and the color specification stored for the color cell is returned in this **XcmsColor** structure.

color_screen_return Returns the color that can be reproduced on the **Screen**.

color_string Specifies the color string.

result_format Specifies the color format for the returned color specifications (*color_screen_return* and *color_exact_return* arguments). If format is XcmsUndefinedFormat and the color string contains a numerical color specification, the specification is returned in the format used in that numerical color specification. If format is XcmsUndefinedFormat and the color string contains a color name, the specification is returned in the format used to store the color in the database.

ncolors Specifies the number of **XcmsColor** structures in the color specification array.

DESCRIPTION

The *XcmsQueryColor()* function obtains the RGB value for the pixel value in the pixel member of the specified **XcmsColor** structure, and then converts the value to the target format as specified by the *result_format* argument. If the pixel is not a valid index into the specified colormap, a BadValue error results. The *XcmsQueryColors()* function obtains the RGB values for pixel values in the pixel members of **XcmsColor** structures, and then converts the values to the target format as specified by the *result_format* argument. If a pixel is not a valid index into the specified colormap, a BadValue error results. If more than one pixel is in error, the one that gets reported is arbitrary.

XcmsQueryColor() and *XcmsQueryColors()* can generate BadColor and BadValue errors.

The *XcmsLookupColor()* function looks up the string name of a color with respect to the screen associated with the specified colormap. It returns both the exact color values and the closest values provided by the screen with respect to the visual type of the specified colormap. The values are returned in the format specified by *result_format*. If the color name is not in the Host Portable Character Encoding the result is implementation dependent. Use of uppercase or lowercase does not matter. *XcmsLookupColor()* returns XcmsSuccess or XcmsSuccessWithCompression if the name is resolved, otherwise it returns XcmsFailure. If XcmsSuccessWithCompression is returned, then the color specification in *color_screen_return* is the result of gamut compression.

DIAGNOSTICS

BadColor

A value for a Colormap argument does not name a defined Colormap.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XcmsAllocColor(), *XcmsStoreColor()*, *XQueryColor()*

Xlib — C Language X Interface

NAME

XcmsSetWhitePoint(), *XcmsSetWhiteAdjustProc()* – modifying CCC attributes

SYNOPSIS

```
Status XcmsSetWhitePoint (XcmsCCC ccc, XcmsColor *color)
XcmsWhiteAdjustProc XcmsSetWhiteAdjustProc (XcmsCCC ccc,
                                             XcmsWhiteAdjustProc white_adjust_proc,
                                             XPointer client_data)
```

ARGUMENTS

ccc Specifies the CCC.
client_data Specifies client data for the white point adjustment procedure or NULL.
color Specifies the new Client White Point.
white_adjust_proc Specifies the white point adjustment procedure.

DESCRIPTION

The *XcmsSetWhitePoint()* function changes the Client White Point in the specified CCC. Note that the pixel member is ignored and that the color specification is left unchanged upon return. The format for the new white point must be *XcmsCIEXYZFormat*, *XcmsCIEuvYFormat*, *XcmsCIExyYFormat*, or *XcmsUndefinedFormat*. If *color* is NULL, this function sets the format component of the Client White Point specification to *XcmsUndefinedFormat()*, indicating that the Client White Point is assumed to be the same as the Screen White Point.

The *XcmsSetWhiteAdjustProc()* function first sets the white point adjustment procedure and client data in the specified CCC with the newly specified procedure and client data and then returns the old procedure.

SEE ALSO

DisplayOfCCC(), *XcmsCCCOFColormap()*, *XcmsConvertColors()*, *XcmsCreateCCC()*, *XcmsDefaultCCC()*

Xlib — C Language X Interface

NAME

XcmsStoreColor(), XcmsStoreColors() – set colors

SYNOPSIS

```
Status XcmsStoreColor (Display *display, Colormap colormap,
                       XcmsColor *color)
Status XcmsStoreColors (Display *display, Colormap colormap,
                       XcmsColor colors[], int ncolors,
                       Bool compression_flags_return[])
```

ARGUMENTS

display Specifies the connection to the X server.

color Specifies the color cell and the color to store. Values specified in this **XcmsColor** structure remain unchanged upon return.

colors Specifies the color specification array of **XcmsColor** structures, each specifying a color cell and the color to store in that cell. Values specified in the array remain unchanged upon return.

colormap Specifies the colormap.

compression_flags_return Specifies an array of Boolean values for returning compression status. If a non-NULL pointer is supplied, each element of the array is set to True if the corresponding color was compressed, and False otherwise. Pass NULL if the compression status is not useful.

ncolors Specifies the number of **XcmsColor** structures in the color specification array.

DESCRIPTION

The *XcmsStoreColor()* function converts the color specified in the **XcmsColor** structure into RGB values and then uses this RGB specification in an **XColor** structure, whose three *flags* (DoRed, DoGreen, and DoBlue) are set, in a call to *XStoreColor()* to change the color cell specified by the pixel member of the **XcmsColor** structure. This pixel value must be a valid index for the specified colormap, and the color cell specified by the pixel value must be a read / write cell. If the pixel value is not a valid index, a BadValue error results. If the color cell is unallocated or is allocated read-only, a BadAccess error results. If the *colormap* is an installed map for its screen, the changes are visible immediately.

Note that *XStoreColor()* has no return value; therefore, a XcmsSuccess return value from this function indicates that the conversion to RGB succeeded and the call to *XStoreColor()* was made. To obtain the actual color stored, use *XcmsQueryColor()*. Due to the screen's hardware limitations or gamut compression, the color stored in the colormap may not be identical to the color specified.

XcmsStoreColor() can generate BadAccess, BadColor, and BadValue errors.

The *XcmsStoreColors()* function converts the colors specified in the array of **XcmsColor** structures into RGB values and then uses these RGB specifications in an **XColor** structures, whose three *flags* (DoRed, DoGreen, and DoBlue) are set, in a call to *XStoreColors()* to change the color cells specified by the pixel member of the corresponding **XcmsColor** structure. Each pixel value must be a valid index for the specified colormap, and the color cell specified by each pixel value must be a read / write cell. If a pixel value is not a valid index, a BadValue error results. If a color cell is

unallocated or is allocated read-only, a `BadAccess` error results. If more than one pixel is in error, the one that gets reported is arbitrary. If the *colormap* is an installed map for its screen, the changes are visible immediately.

Note that `XStoreColors()` has no return value; therefore, a `XcmsSuccess` return value from this function indicates that conversions to RGB succeeded and the call to `XStoreColors()` was made. To obtain the actual colors stored, use `XcmsQueryColors()`. Due to the screen's hardware limitations or gamut compression, the colors stored in the colormap may not be identical to the colors specified.

`XcmsStoreColors()` can generate `BadAccess`, `BadColor`, and `BadValue` errors.

DIAGNOSTICS

`BadAccess`

A client attempted to free a color map entry that it did not already allocate.

`BadAccess`

A client attempted to store into a read-only color map entry.

`BadColor`

A value for a `Colormap` argument does not name a defined `Colormap`.

`BadValue`

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

`XcmsAllocColor()`, `XcmsQueryColor()`

Xlib — C Language X Interface

NAME

XcmsTekHVCQueryMaxC(), XcmsTekHVCQueryMaxV(),
 XcmsTekHVCQueryMaxVC(), XcmsTekHVCQueryMaxVSamples(),
 XcmsTekHVCQueryMinV() – obtain the TekHVC coordinates

SYNOPSIS

```
Status XcmsTekHVCQueryMaxC (XcmsCCC ccc, XcmsFloat hue,
                             XcmsFloat value,
                             XcmsColor *color_return)
Status XcmsTekHVCQueryMaxV (XcmsCCC ccc, XcmsFloat hue,
                             XcmsFloat chroma,
                             XcmsColor *color_return)
Status XcmsTekHVCQueryMaxVC (XcmsCCC ccc, XcmsFloat hue,
                             XcmsColor *color_return)
Status XcmsTekHVCQueryMaxVSamples (XcmsCCC ccc, XcmsFloat hue,
                                    XcmsColor colors_return[],
                                    unsigned int nsamples)
Status XcmsTekHVCQueryMinV (XcmsCCC ccc, XcmsFloat hue,
                             XcmsFloat chroma,
                             XcmsColor *color_return)
```

ARGUMENTS

ccc Specifies the CCC. Note that the CCC's Client White Point and White Point Adjustment procedures are ignored.

chroma Specifies the chroma at which to find maximum Value (MaxV).

colors_in_out
 Returns *nsamples* of color specifications in **XcmsTekHVC** such that the Chroma is the maximum attainable for the Value and Hue. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

color_return
 Returns the maximum Chroma along with the actual Hue and Value (*MaxC*), maximum Value along with the Hue and Chroma (*MaxV*), color specification in XcmsTekHVC for the maximum Chroma, the Value at which that maximum Chroma is reached and actual Hue (*MaxVC*) or minimum Value and the actual Hue and Chroma (*MinL*) at which the maximum Chroma (*MaxC* and *MaxVC*), maximum Value (*MaxV*), or minimum Value (*MinL*) was found. The white point associated with the returned color specification is the Screen White Point. The value returned in the pixel member is undefined.

hue Specifies the Hue in which to find the maximum Chroma (*MaxC* and *MaxVC*), maximum Value (*MaxV*), the maximum Chroma/Value samples (*MaxVSamples*), or the minimum Value (*MinL*).

nsamples
 Specifies the number of samples.

value Specifies the Value in which to find the maximum Chroma (*MaxC*) or minimum Value (*MinL*).

DESCRIPTION

The *XcmsTekHVCQueryMaxC()* function, given a *hue* and *value*, determines the maximum Chroma in TekHVC color space displayable by the screen. Note that it returns the maximum Chroma along with the actual Hue and Value at which the

maximum Chroma was found.

The *XcmsTekHVCQueryMaxV()* function, given a Hue and Chroma, determines the maximum Value in TekHVC color space displayable by the screen. Note that it returns the maximum Value and the actual Hue and Chroma at which the maximum Value was found.

The *XcmsTekHVCQueryMaxVC()* function, given a Hue, determines the maximum Chroma in TekHVC color space displayable by the screen and the Value at which that maximum Chroma is reached. Note that it returns the maximum Chroma, the Value at which that maximum Chroma is reached, and the actual Hue for which the maximum Chroma was found.

The *XcmsTekHVCQueryMaxVSamples()* returns *nsamples* of maximum Value, Chroma at which that maximum Value is reached, and the actual Hue for which the maximum Chroma was found. These sample points may then be used to plot the maximum Value/Chroma boundary of the screen's color gamut for the specified Hue in TekHVC color space.

The *XcmsTekHVCQueryMinV()* function, given a Hue and Chroma, determines the minimum Value in TekHVC color space displayable by the screen. Note that it returns the minimum Value and the actual Hue and Chroma at which the minimum Value was found.

SEE ALSO

XcmsCIELabQueryMaxC(), *XcmsCIELuvQueryMaxC()*, *XcmsQueryBlack()*

Xlib — C Language X Interface

NAME

XmbDrawImageString(), XwcDrawImageString() – draw image text using a single font set

SYNOPSIS

```
void XmbDrawImageString (Display *display, Drawable d,
                        XFontSet font_set, GC gc, int x,
                        int y, char *string, int num_bytes)
void XwcDrawImageString (Display *display, Drawable d,
                        XFontSet font_set, GC gc, int x,
                        int y, wchar_t *string, int num_wchars)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
font_set Specifies the font set.
gc Specifies the GC.
num_bytes Specifies the number of bytes in the *string* argument.
num_wchars Specifies the number of characters in the *string* argument.
string Specifies the character string.
x
y Specify the x and y coordinates.

DESCRIPTION

XmbDrawImageString() and *XwcDrawImageString()* fill a destination rectangle with the background pixel defined in the GC and then paint the text with the foreground pixel. The filled rectangle is the rectangle returned to *overall_logical_return* by *XmbTextExtents()* or *XwcTextExtents()* for the same text and **XFontSet**.

When the **XFontSet** has missing charsets, each unavailable character is drawn with the default string returned by *XCreateFontSet()*. The behavior for an invalid codepoint is undefined.

SEE ALSO

XDrawImageString(), *XDrawString()*, *XDrawText()*, *XmbDrawString()*,
XmbDrawText()

Xlib — C Language X Interface

NAME

XmbDrawString(), XwcDrawString() – draw text using a single font set

SYNOPSIS

```
void XmbDrawString (Display *display, Drawable d,
                   XFontSet font_set, GC gc, int x,
                   int y, char *string, int num_bytes)
void XwcDrawString (Display *display, Drawable d,
                   XFontSet font_set, GC gc, int x,
                   int y, wchar_t *string, int num_wchars)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
font_set Specifies the font set.
gc Specifies the GC.
num_bytes Specifies the number of bytes in the *string* argument.
num_wchars Specifies the number of characters in the *string* argument.
string Specifies the character string.
x
y Specify the x and y coordinates.

DESCRIPTION

XmbDrawString() and *XwcDrawString()* draw the specified text with the foreground pixel. When the **XFontSet** has missing charsets, each unavailable character is drawn with the default string returned by *XCreateFontSet()*. The behavior for an invalid codepoint is undefined.

SEE ALSO

XDrawImageString(), *XDrawString()*, *XDrawText()*, *XmbDrawImageString()*,
XmbDrawText()
Xlib — C Language X Interface

NAME

XmbDrawText(), XwcDrawText() – draw text using multiple font sets

SYNOPSIS

```
void XmbDrawText (Display *display, Drawable d, GC gc,
                  int x, int y, XmbTextItem *items,
                  int nitems)
void XwcDrawText (Display *display, Drawable d, GC gc,
                  int x, int y, XwcTextItem *items,
                  int nitems)
```

ARGUMENTS

d Specifies the drawable.
display Specifies the connection to the X server.
gc Specifies the GC.
items Specifies an array of text items.
nitems Specifies the number of text items in the array.
x
y Specify the x and y coordinates.

DESCRIPTION

XmbDrawText() and *XwcDrawText()* allow complex spacing and font set shifts between text strings. Each text item is processed in turn, with the origin of a text element advanced in the primary draw direction by the escapement of the previous text item. A text item delta specifies an additional escapement of the text item drawing origin in the primary draw direction. A *font_set* member other than None in an item causes the font set to be used for this and subsequent text items in the *text_items* list. Leading text items with *font_set* member set to None will not be drawn.

XmbDrawText() and *XwcDrawText()* do not perform any context-dependent rendering between text segments. Clients may compute the drawing metrics by passing each text segment to *XmbTextExtents()* and *XwcTextExtents()* or *XmbTextPerCharExtents()* and *XwcTextPerCharExtents()*. When the **XFontSet** has missing charsets, each unavailable character is drawn with the default string returned by *XCreateFontSet()*. The behavior for an invalid codepoint is undefined.

STRUCTURES

The **XmbTextItem** structure contains:

```
typedef struct {
    char *chars;           pointer to string
    int nchars;            number of characters
    int delta;             pixel delta between strings
    XFontSet font_set;     fonts, None means don't change
} XmbTextItem;
```

The **XwcTextItem** structure contains:

```
typedef struct {
    wchar_t *chars;        pointer to wide char string
    int nchars;            number of wide characters
    int delta;             pixel delta between strings
    XFontSet font_set;     fonts, None means don't change
}
```

```
    } XwcTextItem;
```

SEE ALSO

XDrawImageString(), *XDrawString()*, *XDrawText()*, *XmbDrawImageString()*,
XmbDrawString()

Xlib — C Language X Interface

NAME

XmbLookupString(), XwcLookupString() – obtain composed input from an input method

SYNOPSIS

```
int XmbLookupString (XIC ic, XKeyPressedEvent *event,
                    char *buffer_return, int bytes_buffer,
                    KeySym *keysym_return, Status *status_return)
int XwcLookupString (XIC ic, XKeyPressedEvent *event,
                    wchar_t *buffer_return, int wchars_buffer,
                    KeySym *keysym_return, Status *status_return)
```

ARGUMENTS

buffer_return
Returns a multibyte string or wide character string (if any) from the input method.

bytes_buffer
wchars_buffer
Specifies space available in return buffer.

event
Specifies the key event to be used.

ic
Specifies the input context.

keysym_return
Returns the KeySym computed from the event if this argument is not NULL.

status_return
Returns a value indicating what kind of data is returned.

DESCRIPTION

The *XmbLookupString()* and *XwcLookupString()* functions return the string from the input method specified in the *buffer_return* argument. If no string is returned, the *buffer_return* argument is unchanged.

The KeySym into which the KeyCode from the event was mapped is returned in the *keysym_return* argument if it is non-NULL and the *status_return* argument indicates that a KeySym was returned. If both a string and a KeySym are returned, the KeySym value does not necessarily correspond to the string returned.

Note that *XmbLookupString()* returns the length of the string in bytes and that *XwcLookupString()* returns the length of the string in characters.

XmbLookupString() and *XwcLookupString()* return text in the encoding of the locale bound to the input method of the specified input context.

Note that each string returned by *XmbLookupString()* and *XwcLookupString()* begins in the initial state of the encoding of the locale (if the encoding of the locale is state-dependent).

Note

In order to insure proper input processing, it is essential that the client pass only *KeyPress()* events to *XmbLookupString()* and *XwcLookupString()*. Their behavior when a client passes a *KeyRelease()* event is undefined.

Clients should check the *status_return* argument before using the other returned values. These two functions both return a value to *status_return* that indicates what has been

returned in the other arguments. The possible values returned are:

- XBufferOverflow()* The input string to be returned is too large for the supplied *buffer_return*. The required size (*XmbLookupString()* in bytes; *XwcLookupString()* in characters) is returned as the value of the function, and the contents of *buffer_return* and *keysym_return* are not modified. The client should recall the function with the same event and a buffer of adequate size in order to obtain the string.
- XLookupNone()* No consistent input has been composed so far. The contents of *buffer_return* and *keysym_return* are not modified, and the function returns zero.
- XLookupChars()* Some input characters have been composed. They are placed in the *buffer_return* argument, and the string length is returned as the value of the function. The string is encoded in the locale bound to the input context. The contents of the *keysym_return* argument is not modified.
- XLookupKeySym()* A KeySym has been returned instead of a string and is returned in *keysym_return*. The contents of the *buffer_return* argument is not modified, and the function returns zero.
- XLookupBoth()* Both a KeySym and a string are returned; *XLookupChars()* and *XLookupKeySym()* occur simultaneously.

It does not make any difference if the input context passed as an argument to *XmbLookupString()* and *XwcLookupString()* is the one currently in possession of the focus or not. Input may have been composed within an input context before it lost the focus, and that input may be returned on subsequent calls to *XmbLookupString()* or *XwcLookupString()*, even though it does not have any more keyboard focus.

SEE ALSO

XLookupKeysym()

Xlib — C Language X Interface

NAME

XmbResetIC(), XwcResetIC() – reset the state of an input context

SYNOPSIS

```
char * XmbResetIC (XIC ic)  
wchar_t * XwcResetIC (XIC ic)
```

ARGUMENTS

ic Specifies the input context.

DESCRIPTION

The *XmbResetIC()* and *XwcResetIC()* functions reset input context to initial state. Any input pending on that context is deleted. Input method is required to clear preedit area, if any, and update status accordingly. Calling *XmbResetIC()* or *XwcResetIC()* does not change the focus.

The return value of *XmbResetIC()* is its current preedit string as a multibyte string. The return value of *XwcResetIC()* is its current preedit string as a wide character string. It is input method implementation dependent whether these routines return a non-NULL string or NULL.

The client should free the returned string by calling *XFree()*.

SEE ALSO

XCreateIC(), *XOpenIM()*, *XSetICFocus()*, *XSetICValues()*

Xlib — C Language X Interface

NAME

XmbTextEscapement(), XwcTextEscapement() – obtain the escapement of text

SYNOPSIS

```
int XmbTextEscapement (XFontSet font_set, char *string,
                      int num_bytes)
int XwcTextEscapement (XFontSet font_set, wchar_t *string,
                      int num_wchars)
```

ARGUMENTS

font_set Specifies the font set.
num_bytes Specifies the number of bytes in the string argument.
num_wchars Specifies the number of characters in the string argument.
string Specifies the character string.

DESCRIPTION

The *XmbTextEscapement()* and *XwcTextEscapement()* functions return the escapement in pixels of the specified string as a value, using the fonts loaded for the specified font set. The escapement is the distance in pixels in the primary draw direction from the drawing origin to the origin of the next character to be drawn, assuming that the rendering of the next character is not dependent on the supplied string.

Regardless of the character rendering order, the escapement is always positive.

SEE ALSO

XmbTextExtents(), *XmbTextPerCharExtents()* *Xlib — C Language X Interface*

NAME

XmbTextExtents(), XwcTextExtents() – compute text extents

SYNOPSIS

```
int XmbTextExtents (XFontSet font_set, char *string,
                   int num_bytes,
                   XRectangle *overall_ink_return,
                   XRectangle *overall_logical_return)
int XwcTextExtents (XFontSet font_set, wchar_t *string,
                   int num_wchars,
                   XRectangle *overall_ink_return,
                   XRectangle *overall_logical_return)
```

ARGUMENTS

font_set Specifies the font set.
num_bytes
 Specifies the number of bytes in the string argument.
num_wchars
 Specifies the number of characters in the string argument.
overall_ink_return
 Returns the overall ink dimensions.
overall_logical_return
 Returns the overall logical dimensions.
string Specifies the character string.

DESCRIPTION

The *XmbTextExtents()* and *XwcTextExtents()* functions set the components of the specified *overall_ink_return* and *overall_logical_return* arguments to the overall bounding box of the string's image, and a logical bounding box for spacing purposes, respectively. They return the value returned by *XmbTextEscapement()* or *XwcTextEscapement()*. These metrics are relative to the drawing origin of the string, using the fonts loaded for the specified font set.

If the *overall_ink_return* argument is non-NULL, it is set to the bounding box of the string's character ink. Note that the *overall_ink_return* for a non-descending horizontally drawn Latin character is conventionally entirely above the baseline, that is, *overall_ink_return.height* $\leq$ $-overall_ink_return.y$. The *overall_ink_return* for a nonkerned character is entirely at and to the right of the origin, that is, *overall_ink_return.x* $\geq$ 0. A character consisting of a single pixel at the origin would set *overall_ink_return* fields *y* = 0, *x* = 0, *width* = 1, *height* = 1.

If the *overall_logical_return* argument is non-NULL, it is set to the bounding box which provides minimum spacing to other graphical features for the string. Other graphical features, for example, a border surrounding the text, should not intersect this rectangle.

When the **XFontSet** has missing charsets, metrics for each unavailable character are taken from the default string returned by *XCreateFontSet()* so that the metrics represent the text as it will actually be drawn. The behavior for an invalid codepoint is undefined.

SEE ALSO

XmbTextEscapement(), *XmbTextPerCharExtents()* *Xlib — C Language X Interface*

NAME

XmbTextListToTextProperty(), XwcTextListToTextProperty(),
 XmbTextPropertyToTextList(), XwcTextPropertyToTextList(), XwcFreeStringList(),
 XDefaultString() – convert text lists and text property structures

SYNOPSIS

```
int XmbTextListToTextProperty (Display *display, char **list,
                              int count, XICCEncodingStyle style,
                              XTextProperty *text_prop_return)
int XwcTextListToTextProperty (Display *display, wchar_t **list,
                              int count, XICCEncodingStyle style,
                              XTextProperty *text_prop_return)
int XmbTextPropertyToTextList (Display *display,
                              XTextProperty *text_prop,
                              char ***list_return,
                              int *count_return)
int XwcTextPropertyToTextList (Display *display,
                              XTextProperty *text_prop,
                              wchar_t ***list_return,
                              int *count_return)
void XwcFreeStringList (wchar_t **list)
char *XDefaultString (void)
```

ARGUMENTS

display Specifies the connection to the X server.
list Specifies a list of null-terminated character strings.
count Specifies the number of strings specified.
style Specifies the manner in which the property is encoded.
text_prop_return
 Returns the **XTextProperty** structure.
text_prop
 Specifies the **XTextProperty** structure to be used.
list_return
 Returns a list of null-terminated character strings.
count_return
 Returns the number of strings.
list Specifies the list of strings to be freed.

DESCRIPTION

The *XmbTextListToTextProperty()* and *XwcTextListToTextProperty()* functions set the specified **XTextProperty** value to a set of null-separated elements representing the concatenation of the specified list of null-terminated text strings. A final terminating null is stored at the end of the value field of *text_prop_return* but is not included in the *nitems* member.

The functions set the encoding field of *text_prop_return* to an Atom for the specified display naming the encoding determined by the specified style, and convert the specified text list to this encoding for storage in the *text_prop_return* value field. If the style **XStringStyle** or **XCompoundTextStyle** is specified, this encoding is “STRING” or “COMPOUND_TEXT”, respectively. If the style **XTextStyle** is specified, this encoding is the encoding of the current locale. If the style **XStdICCTextStyle** is specified, this

encoding is “STRING” if the text is fully convertible to STRING, else “COMPOUND_TEXT”.

If insufficient memory is available for the new value string, the functions return XNoMemory. If the current locale is not supported, the functions return XLocaleNotSupported. In both of these error cases, the functions do not set *text_prop_return*.

To determine if the functions are guaranteed not to return XLocaleNotSupported, use *XSupportsLocale()*.

If the supplied text is not fully convertible to the specified encoding, the functions return the number of unconvertible characters. Each unconvertible character is converted to an implementation-defined and encoding-specific default string. Otherwise, the functions return *Success()*. Note that full convertibility to all styles except XStringStyle is guaranteed.

To free the storage for the value field, use *XFree()*.

The *XmbTextPropertyToTextList()* and *XwcTextPropertyToTextList()* functions return a list of text strings in the current locale representing the null-separated elements of the specified **XTextProperty** structure. The data in *text_prop* must be format 8.

Multiple elements of the property (for example, the strings in a disjoint text selection) are separated by a null byte. The contents of the property are not required to be null-terminated; any terminating null should not be included in *text_prop.nitems*.

If insufficient memory is available for the list and its elements, *XmbTextPropertyToTextList()* and *XwcTextPropertyToTextList()* return XNoMemory. If the current locale is not supported, the functions return XLocaleNotSupported. Otherwise, if the encoding field of *text_prop* is not convertible to the encoding of the current locale, the functions return XConverterNotFound. For supported locales, existence of a converter from COMPOUND_TEXT, STRING, or the encoding of the current locale is guaranteed if *XSupportsLocale()* returns True for the current locale (but the actual text may contain unconvertible characters.) Conversion of other encodings is implementation-dependent. In all of these error cases, the functions do not set any return values.

Otherwise, *XmbTextPropertyToTextList()* and *XwcTextPropertyToTextList()* return the list of null-terminated text strings to *list_return*, and the number of text strings to *count_return*.

If the value field of *text_prop* is not fully convertible to the encoding of the current locale, the functions return the number of unconvertible characters. Each unconvertible character is converted to a string in the current locale that is specific to the current locale. To obtain the value of this string, use *XDefaultString()*. Otherwise, *XmbTextPropertyToTextList()* and *XwcTextPropertyToTextList()* return *Success*.

To free the storage for the list and its contents returned by *XmbTextPropertyToTextList()*, use *XFreeStringList()*. To free the storage for the list and its contents returned by *XwcTextPropertyToTextList()*, use *XwcFreeStringList()*.

The *XwcFreeStringList()* function frees memory allocated by *XwcTextPropertyToTextList()*.

The *XDefaultString()* function returns the default string used by Xlib for text conversion (for example, in *XmbTextListToTextProperty()*). The default string is the string in the

current locale which is output when an unconvertible character is found during text conversion. If the string returned by *XDefaultString()* is the empty string (""), no character is output in the converted text. *XDefaultString()* does not return NULL.

The string returned by *XDefaultString()* is independent of the default string for text drawing; see *XCreateFontSet()* to obtain the default string for an *XFontSet()*.

The behavior when an invalid codepoint is supplied to any Xlib function is undefined.

The returned string is null-terminated. It is owned by Xlib and should not be modified or freed by the client. It may be freed after the current locale is changed. Until freed, it will not be modified by Xlib.

STRUCTURES

The **XTextProperty** structure contains:

```
typedef struct
{
    unsigned char *value;    property data
    Atom encoding;          type of property
    int format;              8, 16, or 32
    unsigned long nitems;    number of items in value
} XTextProperty;
```

The **XICCEncodingStyle** structure contains:

```
#define XNoMemory          -1
#define XLocaleNotSupported -2
#define XConverterNotFound -3

typedef enum {
    XStringStyle,          STRING
    XCompoundTextStyle,    COMPOUND_TEXT
    XTextStyle,            text in owner's encoding (current locale)
    XStdICCTextStyle       STRING, else COMPOUND_TEXT
} XICCEncodingStyle;
```

SEE ALSO

XSetTextProperty(), *XStringListToTextProperty()*

Xlib — C Language X Interface

NAME

XmbTextPerCharExtents(), XwcTextPerCharExtents() – obtain per-character information for a text string

SYNOPSIS

```
Status XmbTextPerCharExtents (XFontSet font_set, char *string,
                               int num_bytes,
                               XRectangle *ink_array_return,
                               XRectangle *logical_array_return,
                               int array_size, int *num_chars_return,
                               XRectangle *overall_ink_return,
                               XRectangle *overall_logical_return)

Status XwcTextPerCharExtents (XFontSet font_set, wchar_t *string,
                              int num_wchars,
                              XRectangle *ink_array_return,
                              XRectangle *logical_array_return,
                              int array_size, int *num_chars_return,
                              XRectangle *overall_ink_return,
                              XRectangle *overall_logical_return)
```

ARGUMENTS

array_size Specifies the size of *ink_array_return* and *logical_array_return*. Note that the caller must pass in arrays of this size.

font_set Specifies the font set.

ink_array_return Returns the ink dimensions for each character.

logical_array_return Returns the logical dimensions for each character.

num_bytes Specifies the number of bytes in the *string* argument.

num_chars_return Returns the number characters in the *string* argument.

num_wchars Specifies the number of characters in the *string* argument.

overall_ink_return Returns the overall ink extents of the entire *string*.

overall_logical_return Returns the overall logical extents of the entire *string*.

string Specifies the character string.

DESCRIPTION

The *XmbTextPerCharExtents()* and *XwcTextPerCharExtents()* return the text dimensions of each character of the specified text, using the fonts loaded for the specified font set. Each successive element of *ink_array_return* and *logical_array_return* is set to the successive character's drawn metrics, relative to the drawing origin of the string, one **XRectangle** for each character in the supplied text string. The number of elements of *ink_array_return* and *logical_array_return* that have been set is returned to *num_chars_return*.

Each element of *ink_array_return* is set to the bounding box of the corresponding character's drawn foreground color. Each element of *logical_array_return* is set to the bounding box which provides minimum spacing to other graphical features for the corresponding character. Other graphical features should not intersect any of the *logical_array_return* rectangles.

Note that an **XRectangle** represents the effective drawing dimensions of the character, regardless of the number of font glyphs that are used to draw the character, or the direction in which the character is drawn. If multiple characters map to a single character glyph, the dimensions of all the **XRectangles** of those characters are the same.

When the **XFontSet** has missing charsets, metrics for each unavailable character are taken from the default string returned by *XCreateFontSet()*, so that the metrics represent the text as it will actually be drawn. The behavior for an invalid codepoint is undefined.

If the *array_size* is too small for the number of characters in the supplied text, the functions return zero and *num_chars_return* is set to the number of rectangles required. Otherwise, the routines return a non-zero value.

If the *overall_ink_return* or *overall_logical_return* argument is non-NULL, *XmbTextPerCharExtents()* and *XwcTextPerCharExtents()* return the maximum extent of the string's metrics to *overall_ink_return* or *overall_logical_return*, as returned by *XmbTextExtents()* or *XwcTextExtents()*.

SEE ALSO

XmbTextEscapement(), *XmbTextExtents()* *Xlib* — C Language X Interface

NAME

XrmEnumerateDatabase() – enumerate resource database entries

SYNOPSIS

```
#define      XrmEnumAllLevels      0
#define      XrmEnumOneLevel      1

Bool XrmEnumerateDatabase (XrmDatabase database,
                           XrmNameList name_prefix,
                           XrmClassList class_prefix,
                           int mode, Bool (*proc)(),
                           XPointer arg)
```

ARGUMENTS

database Specifies the resource database.
name_prefix Specifies the resource name prefix.
class_prefix Specifies the resource class prefix.
mode Specifies the number of levels to enumerate.
proc Specifies the procedure that is to be called for each matching entry.
arg Specifies the user-supplied argument that will be passed to the procedure.

DESCRIPTION

The *XrmEnumerateDatabase()* function calls the specified procedure for each resource in the database that would match some completion of the given name/class resource prefix. The order in which resources are found is implementation-dependent. If mode is *XrmEnumOneLevel*, then a resource must match the given name/class prefix with just a single name and class appended. If mode is *XrmEnumAllLevels*, the resource must match the given name/class prefix with one or more names and classes appended. If the procedure returns True, the enumeration terminates and the function returns True. If the procedure always returns False, all matching resources are enumerated and the function returns False.

The procedure is called with the following arguments:

```
(*proc )( database, bindings, quarks, type, value, arg )
XrmDatabase *database ;
XrmBindingList bindings ;
XrmQuarkList quarks ;
XrmRepresentation *type ;
XrmValue *value ;
XPointer closure ;
```

The bindings and quarks lists are terminated by NULLQUARK. Note that pointers to the database and type are passed, but these values should not be modified.

SEE ALSO

XrmGetResource(), *XrmInitialize()*, *XrmPutResource()*

Xlib — C Language X Interface

NAME

XrmGetFileDatabase(), XrmPutFileDatabase(), XrmGetStringDatabase(),
 XrmLocaleOfDatabase(), XrmGetDatabase(), XrmSetDatabase(), XrmDestroyDatabase()
 – retrieve and store resource databases

SYNOPSIS

```
XrmDatabase XrmGetFileDatabase (char *filename)
void XrmPutFileDatabase (XrmDatabase database, char *stored_db)
XrmDatabase XrmGetStringDatabase (char *data)
char *XrmLocaleOfDatabase (XrmDatabase database)
XrmDatabase XrmGetDatabase (Display *display)
void XrmSetDatabase (Display *display, XrmDatabase database)
void XrmDestroyDatabase (XrmDatabase database)
```

ARGUMENTS

filename Specifies the resource database file name.
database Specifies the database that is to be used.
stored_db
 Specifies the file name for the stored database.
data Specifies the database contents using a string.
database Specifies the resource database.
display Specifies the connection to the X server.

DESCRIPTION

The *XrmGetFileDatabase()* function opens the specified file, creates a new resource database, and loads it with the specifications read in from the specified file. The specified file must contain a sequence of entries in valid ResourceLine format (see section 15.1). The file is parsed in the current locale, and the database is created in the current locale. If it cannot open the specified file, *XrmGetFileDatabase()* returns NULL.

The *XrmPutFileDatabase()* function stores a copy of the specified database in the specified file. Text is written to the file as a sequence of entries in valid ResourceLine format (see section 15.1). The file is written in the locale of the database. Entries containing resource names that are not in the Host Portable Character Encoding, or containing values that are not in the encoding of the database locale, are written in an implementation-dependent manner. The order in which entries are written is implementation dependent. Entries with representation types other than “String” are ignored.

The *XrmGetStringDatabase()* function creates a new database and stores the resources specified in the specified null-terminated string. *XrmGetStringDatabase()* is similar to *XrmGetFileDatabase()* except that it reads the information out of a string instead of out of a file. The string must contain a sequence of entries in valid ResourceLine format (see section 15.1). The string is parsed in the current locale, and the database is created in the current locale.

If database is NULL, *XrmDestroyDatabase()* returns immediately.

The *XrmLocaleOfDatabase()* function returns the name of the locale bound to the specified database, as a null-terminated string. The returned locale name string is owned by Xlib and should not be modified or freed by the client. Xlib is not permitted to free the string until the database is destroyed. Until the string is freed, it will not be modified by Xlib.

The *XrmGetDatabase()* function returns the database associated with the specified display. It returns NULL if a database has not yet been set.

The *XrmSetDatabase()* function associates the specified resource database (or NULL) with the specified display. The database previously associated with the display (if any) is not destroyed. A client or toolkit may find this function convenient for retaining a database once it is constructed.

FILE SYNOPSIS

The syntax of a resource file is a sequence of resource lines terminated by newline characters or end of file. The syntax of an individual resource line is:

```
ResourceLine    = Comment | IncludeFile | ResourceSpec |
                  <empty line>
Comment         = "!" {<any character except null or newline>}
IncludeFile     = "#" WhiteSpace "include" WhiteSpace
                  FileName WhiteSpace
FileName        = <valid filename for operating system>
ResourceSpec    = WhiteSpace ResourceName WhiteSpace ":"
                  WhiteSpace Value
ResourceName    = [Binding] {Component Binding} ComponentName
Binding         = "." | "*"
WhiteSpace      = {<space> | <horizontal tab>}
Component       = "?" | ComponentName
ComponentName   = NameChar {NameChar}
NameChar       = "a"-"z" | "A"-"Z" | "0"-"9" | "_" | "-"
Value          = T{= {<any character except null or unescaped newline>}
T}
```

Elements separated by vertical bar (|) are alternatives. Curly braces ({...}) indicate zero or more repetitions of the enclosed elements. Square brackets ([...]) indicate that the enclosed element is optional. Quotes ("...") are used around literal characters.

IncludeFile lines are interpreted by replacing the line with the contents of the specified file. The word "include" must be in lowercase. The filename is interpreted relative to the directory of the file in which the line occurs (for example, if the filename contains no directory or contains a relative directory specification).

If a ResourceName contains a contiguous sequence of two or more Binding characters, the sequence will be replaced with single "." character if the sequence contains only "." characters, otherwise the sequence will be replaced with a single "*" character.

A resource database never contains more than one entry for a given ResourceName. If a resource file contains multiple lines with the same ResourceName, the last line in the file is used.

Any whitespace character before or after the name or colon in a ResourceSpec are ignored. To allow a Value to begin with whitespace, the two-character sequence "\space" (backslash followed by space) is recognized and replaced by a space character, and the two-character sequence "\tab" (backslash followed by horizontal tab) is recognized and replaced by a horizontal tab character. To allow a Value to contain embedded newline characters, the two-character sequence "\n" is recognized and replaced by a newline character. To allow a Value to be broken across multiple lines in a text file, the two-character sequence "\newline" (backslash followed by newline) is

recognized and removed from the value. To allow a Value to contain arbitrary character codes, the four-character sequence “\nnn”, where each *n* is a digit character in the range of “0”–“7”, is recognized and replaced with a single byte that contains the octal value specified by the sequence. Finally, the two-character sequence “\ ” is recognized and replaced with a single backslash.

SEE ALSO

XrmGetResource(), *XrmInitialize()*, *XrmPutResource()*

Xlib — C Language X Interface

NAME

XrmGetResource(), XrmQGetResource(), XrmQGetSearchList(),
XrmQGetSearchResource() – retrieve database resources and search lists

SYNOPSIS

```

Bool XrmGetResource (XrmDatabase database, char *str_name,
                    char *str_class, char **str_type_return,
                    XrmValue *value_return)
Bool XrmQGetResource (XrmDatabase database, XrmNameList quark_name,
                    XrmClassList quark_class,
                    XrmRepresentation *quark_type_return,
                    XrmValue *value_return)
typedef XrmHashTable *XrmSearchList
Bool XrmQGetSearchList (XrmDatabase database, XrmNameList names,
                    XrmClassList classes,
                    XrmSearchList list_return,
                    int list_length)
Bool XrmQGetSearchResource (XrmSearchList list, XrmName name,
                    XrmClass class,
                    XrmRepresentation *type_return,
                    XrmValue *value_return)

```

ARGUMENTS

class Specifies the resource class.

classes Specifies a list of resource classes.

database Specifies the database that is to be used.

list Specifies the search list returned by *XrmQGetSearchList()*.

list_length Specifies the number of entries (not the byte size) allocated for *list_return*.

list_return Returns a search list for further use.

name Specifies the resource name.

names Specifies a list of resource names.

quark_class Specifies the fully qualified class of the value being retrieved (as a quark).

quark_name Specifies the fully qualified name of the value being retrieved (as a quark).

quark_type_return Returns the representation type of the destination (as a quark).

str_class Specifies the fully qualified class of the value being retrieved (as a string).

str_name Specifies the fully qualified name of the value being retrieved (as a string).

str_type_return Returns the representation type of the destination (as a string).

type_return Returns data representation type.

value_return Returns the value in the database.

DESCRIPTION

The *XrmGetResource()* and *XrmQGetResource()* functions retrieve a resource from the specified database. Both take a fully qualified name/class pair, a destination resource representation, and the address of a value (size/address pair). The value and returned type point into database memory; therefore, you must not modify the data.

The database only frees or overwrites entries on *XrmPutResource()*, *XrmQPutResource()*, or *XrmMergeDatabases()*. A client that is not storing new values into the database or is not merging the database should be safe using the address passed back at any time until it exits. If a resource was found, both *XrmGetResource()* and *XrmQGetResource()* return True; otherwise, they return False.

The *XrmQGetSearchList()* function takes a list of names and classes and returns a list of database levels where a match might occur. The returned list is in best-to-worst order and uses the same algorithm as *XrmGetResource()* for determining precedence. If *list_return* was large enough for the search list, *XrmQGetSearchList()* returns True; otherwise, it returns False.

The size of the search list that the caller must allocate is dependent upon the number of levels and wildcards in the resource specifiers that are stored in the database. The worst case length is 3^n , where n is the number of name or class components in names or classes.

When using *XrmQGetSearchList()* followed by multiple probes for resources with a common name and class prefix, only the common prefix should be specified in the name and class list to *XrmQGetSearchList()*.

The *XrmQGetSearchResource()* function searches the specified database levels for the resource that is fully identified by the specified name and class. The search stops with the first match. *XrmQGetSearchResource()* returns True if the resource was found; otherwise, it returns False.

A call to *XrmQGetSearchList()* with a name and class list containing all but the last component of a resource name followed by a call to *XrmQGetSearchResource()* with the last component name and class returns the same database entry as *XrmGetResource()* and *XrmQGetResource()* with the fully qualified name and class.

MATCHING RULES

The algorithm for determining which resource database entry matches a given query is the heart of the resource manager. All queries must fully specify the name and class of the desired resource (use of "*" and "?" are not permitted). The library supports up to 100 components in a full name or class. Resources are stored in the database with only partially specified names and classes, using pattern matching constructs. An asterisk (*) is a loose binding and is used to represent any number of intervening components, including none. A period (.) is a tight binding and is used to separate immediately adjacent components. A question mark (?) is used to match any single component name or class. A database entry cannot end in a loose binding; the final component (which cannot be "?") must be specified. The lookup algorithm searches the database for the entry that most closely matches (is most specific for) the full name and class being queried. When more than one database entry matches the full name and class, precedence rules are used to select just one.

The full name and class are scanned from left to right (from highest level in the hierarchy to lowest), one component at a time. At each level, the corresponding component and/or binding of each matching entry is determined, and these matching

components and bindings are compared according to precedence rules. Each of the rules is applied at each level, before moving to the next level, until a rule selects a single entry over all others. The rules (in order of precedence) are:

1. An entry that contains a matching component (whether name, class, or "?") takes precedence over entries that elide the level (that is, entries that match the level in a loose binding).
2. An entry with a matching name takes precedence over both entries with a matching class and entries that match using "?". An entry with a matching class takes precedence over entries that match using "?".
3. An entry preceded by a tight binding takes precedence over entries preceded by a loose binding.

SEE ALSO

XrmInitialize(), *XrmMergeDatabases()*, *XrmPutResource()*, *XrmUniqueQuark()*

Xlib — C Language X Interface

NAME

XrmInitialize(), XrmParseCommand(), XrmValue(), XrmOptionKind(),
XrmOptionDescRec() – initialize the Resource Manager, Resource Manager structures,
and parse the command line

SYNOPSIS

```
void XrmInitialize (void)
void XrmParseCommand (XrmDatabase *database, XrmOptionDescList table,
                     int table_count, char *name,
                     int *argc_in_out, char **argv_in_out)
```

ARGUMENTS

argc_in_out Specifies the number of arguments and returns the number of remaining arguments.

argv_in_out Specifies the command line arguments and returns the remaining arguments.

database Specifies the resource database.

name Specifies the application name.

table Specifies the table of command line arguments to be parsed.

table_count Specifies the number of entries in the table.

DESCRIPTION

The *XrmInitialize()* function initialize the resource manager. It must be called before any other Xrm functions are used.

The *XrmParseCommand()* function parses an (argc, argv) pair according to the specified option table, loads recognized options into the specified database with type “String,” and modifies the (argc, argv) pair to remove all recognized options. If database contains NULL, *XrmParseCommand()* creates a new database and returns a pointer to it. Otherwise, entries are added to the database specified. If a database is created, it is created in the current locale.

The specified table is used to parse the command line. Recognized options in the table are removed from argv, and entries are added to the specified resource database. The table entries contain information on the option string, the option name, the style of option, and a value to provide if the option kind is XrmoptionNoArg. The option names are compared byte-for-byte to arguments in argv, independent of any locale. The resource values given in the table are stored in the resource database without modification. All resource database entries are created using a “String” representation type. The argc argument specifies the number of arguments in argv and is set on return to the remaining number of arguments that were not parsed. The name argument should be the name of your application for use in building the database entry. The name argument is prefixed to the resourceName in the option table before storing a database entry. No separating (binding) character is inserted, so the table must contain either a period (.) or an asterisk (*) as the first character in each resourceName entry. To specify a more completely qualified resource name, the resourceName entry can contain multiple components. If the name argument and the resourceNames are not in the Host Portable Character Encoding the result is implementation dependent.

STRUCTURES

The **XrmValue**, **XrmOptionKind**, and **XrmOptionDescRec** structures contain:

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

typedef enum {
    XrmoptionNoArg,           Value is specified in XrmOptionDescRec.value
    XrmoptionIsArg,          Value is the option string itself
    XrmoptionStickyArg,      Value is characters immediately following option
    XrmoptionSepArg,         Value is next argument in argv
    XrmoptionResArg,         Resource and value in next argument in argv
    XrmoptionSkipArg,        Ignore this option and the next argument in argv
    XrmoptionSkipLine,       Ignore this option and the rest of argv
    XrmoptionSkipNArgs       Ignore this option and the next
                             XrmOptionDescRec.value arguments in argv
} XrmOptionKind;

typedef struct {
    char *option;             Option specification string in argv
    char *specifier;          Binding and resource name (sans application name)
    XrmOptionKind argKind;    Which style of option it is
    XPointer value;           Value to provide if XrmoptionNoArg or
                             XrmoptionSkipNArgs
} XrmOptionDescRec, *XrmOptionDescList;
```

SEE ALSO

XrmGetResource(), *XrmMergeDatabases()*, *XrmPutResource()*, *XrmUniqueQuark()*

Xlib — C Language X Interface

NAME

XrmMergeDatabases(), XrmCombineDatabase(), XrmCombineFileDatabase() – merge resource databases

SYNOPSIS

```
void XrmMergeDatabases (XrmDatabase source_db, *target_db)
void XrmCombineDatabase (XrmDatabase source_db, *target_db,
                        Bool override)
void XrmCombineFileDatabase (char *filename,
                            XrmDatabase *target_db,
                            Bool override)
```

ARGUMENTS

source_db

Specifies the resource database that is to be merged into the target database.

target_db

Specifies the resource database into which the source database is to be merged.

filename Specifies the resource database file name.

DESCRIPTION

The *XrmMergeDatabases()* function merges the contents of one database into another. If the same specifier is used for an entry in both databases, the entry in the *source_db* will replace the entry in the *target_db* (that is, it overrides *target_db*). If *target_db* contains NULL, *XrmMergeDatabases()* simply stores *source_db* in it. Otherwise, *source_db* is destroyed by the merge, but the database pointed to by *target_db* is not destroyed. The database entries are merged without changing values or types, regardless of the locales of the databases. The locale of the target database is not modified.

The *XrmCombineDatabase()* function merges the contents of one database into another. If the same specifier is used for an entry in both databases, the entry in the *source_db* will replace the entry in the *target_db* if *override* is True; otherwise, the entry in *source_db* is discarded. If *target_db* contains NULL, *XrmCombineDatabase()* simply stores *source_db* in it. Otherwise, *source_db* is destroyed by the merge, but the database pointed to by *target_db* is not destroyed. The database entries are merged without changing values or types, regardless of the locales of the databases. The locale of the target database is not modified.

The *XrmCombineFileDatabase()* function merges the contents of a resource file into a database. If the same specifier is used for an entry in both the file and the database, the entry in the file will replace the entry in the database if *override* is True; otherwise, the entry in file is discarded. The file is parsed in the current locale. If the file cannot be read a zero status is returned; otherwise a nonzero status is returned. If *target_db* contains NULL, *XrmCombineFileDatabase()* creates and returns a new database to it. Otherwise, the database pointed to by *target_db* is not destroyed by the merge. The database entries are merged without changing values or types, regardless of the locale of the database. The locale of the target database is not modified.

SEE ALSO

XrmGetResource(), *XrmInitialize()*, *XrmPutResource()*

Xlib — C Language X Interface

NAME

XrmPutResource(), XrmQPutResource(), XrmPutStringResource(),
XrmQPutStringResource(), XrmPutLineResource() – store database resources

SYNOPSIS

```
void XrmPutResource (XrmDatabase *database, char *specifier,
                    char *type, XrmValue *value)
void XrmQPutResource (XrmDatabase *database,
                    XrmBindingList bindings,
                    XrmQuarkList quarks,
                    XrmRepresentation type,
                    XrmValue *value)
void XrmPutStringResource (XrmDatabase *database, char *specifier,
                          char *value)
void XrmQPutStringResource (XrmDatabase *database,
                          XrmBindingList bindings,
                          XrmQuarkList quarks, char *value)
void XrmPutLineResource (XrmDatabase *database, char *line)
```

ARGUMENTS

bindings Specifies a list of bindings.
database Specifies the resource database.
line Specifies the resource name and value pair as a single string.
quarks Specifies the complete or partial name or the class list of the resource.
specifier Specifies a complete or partial specification of the resource.
type Specifies the type of the resource.
value Specifies the value of the resource, which is specified as a string.

DESCRIPTION

If database contains NULL, *XrmPutResource()* creates a new database and returns a pointer to it. *XrmPutResource()* is a convenience function that calls *XrmStringToBindingQuarkList()* followed by:

```
XrmQPutResource(database, bindings, quarks, XrmStringToQuark(type), value)
```

If the *specifier* and *type* are not in the Host Portable Character Encoding the result is implementation dependent. The *value* is stored in the database without modification.

If database contains NULL, *XrmQPutResource()* creates a new database and returns a pointer to it. If a resource entry with the identical bindings and quarks already exists in the database, the previous value is replaced by the new specified value. The *value* is stored in the database without modification.

If database contains NULL, *XrmPutStringResource()* creates a new database and returns a pointer to it. *XrmPutStringResource()* adds a resource with the specified value to the specified database. *XrmPutStringResource()* is a convenience function that first calls *XrmStringToBindingQuarkList()* on the specifier and then calls *XrmQPutResource()*, using a “String” representation type. If the *specifier* is not in the Host Portable Character Encoding the result is implementation dependent. The *value* is stored in the database without modification.

If database contains NULL, *XrmQPutStringResource* creates a new database and returns a pointer to it. *XrmQPutStringResource* is a convenience routine that constructs an

XrmValue for the value string (by calling *strlen()* to compute the size) and then calls *XrmQPutResource()*, using a “String” representation type. The value is stored in the database without modification.

If database contains NULL, *XrmPutLineResource()* creates a new database and returns a pointer to it. *XrmPutLineResource()* adds a single resource entry to the specified database. The *line* must be in valid ResourceLine format (see section 15.1). The string is parsed in the locale of the database. If the ResourceName is not in the Host Portable Character Encoding the result is implementation dependent. Note that comment lines are not stored.

SEE ALSO

XrmGetResource(), *XrmInitialize()*, *XrmMergeDatabases()*, *XrmUniqueQuark()*

Xlib — C Language X Interface

NAME

XrmUniqueQuark(), XrmStringToQuark(), XrmPermStringToQuark(),
XrmQuarkToString(), XrmStringToQuarkList(), XrmStringToBindingQuarkList() –
manipulate resource quarks

SYNOPSIS

```
XrmQuark XrmUniqueQuark (void)

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

XrmQuark XrmStringToQuark (char *string)

XrmQuark XrmPermStringToQuark (char *string)

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

XrmQuark XrmStringToQuark (char *string)

XrmQuark XrmPermStringToQuark (char *string)
#define XrmNameToString(name) XrmQuarkToString(name)
#define XrmClassToString(class) XrmQuarkToString(class)
#define XrmRepresentationToString(type) XrmQuarkToString(type)

char *XrmQuarkToString (XrmQuark quark)
#define XrmStringToNameList(str, name) XrmStringToQuarkList((str), (name))
#define XrmStringToClassList(str, class) XrmStringToQuarkList((str), (class))

void XrmStringToQuarkList (char *string,
                          XrmQuarkList quarks_return)
XrmStringToBindingQuarkList (char *string,
                            XrmBindingList bindings_return,
                            XrmQuarkList quarks_return)
```

ARGUMENTS

bindings_return Returns the binding list.

quark Specifies the quark for which the equivalent string is desired.

quarks_return Returns the list of quarks.

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

DESCRIPTION

The *XrmUniqueQuark()* function allocates a quark that is guaranteed not to represent any string that is known to the resource manager.

These functions can be used to convert from string to quark representation. If the string is not in the Host Portable Character Encoding the conversion is implementation dependent. The string argument to *XrmStringToQuark()* need not be permanently allocated storage. *XrmPermStringToQuark()* is just like *XrmStringToQuark()*, except that Xlib is permitted to assume the string argument is permanently allocated, and hence that it can be used as the value to be returned by *XrmQuarkToString()*.

This function can be used to convert from quark representation to string. The string pointed to by the return value must not be modified or freed. The returned string is byte-for-byte equal to the original string passed to one of the string-to-quark routines. If no string exists for that quark, *XrmQuarkToString()* returns NULL. For any given quark, if *XrmQuarkToString()* returns a non-NULL value, all future calls will return the same value (identical address).

These functions can be used to convert from string to quark representation. If the string is not in the Host Portable Character Encoding the conversion is implementation dependent. The string argument to *XrmStringToQuark()* need not be permanently allocated storage. *XrmPermStringToQuark()* is just like *XrmStringToQuark()*, except that Xlib is permitted to assume the string argument is permanently allocated, and hence that it can be used as the value to be returned by *XrmQuarkToString()*.

The *XrmStringToQuarkList()* function converts the null-terminated string (generally a fully qualified name) to a list of quarks. Note that the string must be in the valid ResourceName format (see section 15.1). If the string is not in the Host Portable Character Encoding the conversion is implementation dependent.

A binding list is a list of type **XrmBindingList** and indicates if components of name or class lists are bound tightly or loosely (that is, if wildcarding of intermediate components is specified).

```
typedef enum {XrmBindTightly, XrmBindLoosely}
             XrmBinding, *XrmBindingList;
```

XrmBindTightly() indicates that a period separates the components, and *XrmBindLoosely()* indicates that an asterisk separates the components.

The *XrmStringToBindingQuarkList()* function converts the specified string to a binding list and a quark list. If the string is not in the Host Portable Character Encoding the conversion is implementation dependent. Component names in the list are separated by a period or an asterisk character. If the string does not start with period or asterisk, a period is assumed. For example, “*a.b*c” becomes:

quarks	a	b	c
bindings	loose	tight	loose

SEE ALSO

XrmGetResource(), *XrmInitialize()*, *XrmMergeDatabases()*, *XrmPutResource()*

Xlib — C Language X Interface

X Intrinsic Library

The routines in this chapter comprise the X Intrinsic Library.

To use these routines, you must include the header file **</usr/X11/include/X11/XI.h>**.

NAME

XAllowDeviceEvents() – release queued events

SYNOPSIS

```
XAllowDeviceEvents (Display *display, XDevice *device,
                    int event_mode, Time time)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device from which events are to be allowed.

event_mode Specifies the event mode. You can pass AsyncThisDevice, SyncThisDevice, ReplayThisDevice, AsyncOtherDevices, SyncAll, or AsyncAll.

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

DESCRIPTION

The *XAllowDeviceEvents()* function releases some queued events if the client has caused a device to freeze. It has no effect if the specified *time* is earlier than the last-grab time of the most recent active grab for the client and device, or if the specified time is later than the current X server time.

The following describes the processing that occurs depending on what constant you pass to the *event_mode* argument.

AsyncThisDevice

If the specified device is frozen by the client, event processing for that device continues as usual. If the device is frozen multiple times by the client on behalf of multiple separate grabs, AsyncThisDevice thaws for all. AsyncThisDevice has no effect if the specified device is not frozen by the client, but the device need not be grabbed by the client.

SyncThisDevice

If the specified device is frozen and actively grabbed by the client, event processing for that device continues normally until the next key or button event is reported to the client. At this time, the specified device again appears to freeze. However, if the reported event causes the grab to be released, the specified device does not freeze. *SyncThisDevice* has no effect if the specified device is not frozen by the client or is not grabbed by the client.

ReplayThisDevice

If the specified device is actively grabbed by the client and is frozen as the result of an event having been sent to the client (either from the activation of a GrabDeviceButton or from a previous AllowDeviceEvents with mode SyncThisDevice, but not from a GrabDevice), the grab is released and that event is completely reprocessed. This time, however, the request ignores any passive grabs at or above (toward the root) that the grab-window of the grab just released. The request has no effect if the specified device is not grabbed by the client or if it is not frozen as the result of an event.

AsyncOtherDevices

If the remaining devices are frozen by the client, event processing for them continues as usual. If the other devices are frozen multiple times by the client on behalf of multiple grabs, AsyncOtherDevices "thaws" for all.

AsyncOtherDevices has no effect if the devices are not frozen by the client.

SyncAll If all devices are frozen by the client, event processing (for all devices)

continues normally until the next button or key event is reported to the client for a grabbed device, at which time all devices again appear to freeze. However, if the reported event causes the grab to be released, then the devices do not freeze. If any device is still grabbed, then a subsequent event for it will still cause all devices to freeze. SyncAll has no effect unless all devices are frozen by the client. If any device is frozen twice by the client on behalf of two separate grabs, SyncAll thaws for both. A subsequent freeze for SyncAll will only freeze each device once.

AsyncAll

If all devices are frozen by the client, event processing for all devices continues normally. If any device is frozen multiple times by the client on behalf of multiple separate grabs, AsyncAll thaws for all. AsyncAll has no effect unless all devices are frozen by the client.

AsyncThisDevice, SyncThisDevice, and ReplayThisDevice have no effect on the processing of events from the remaining devices. AsyncOtherDevices has no effect on the processing of events from the specified device. When the *event_mode* is SyncAll or AsyncAll, the device parameter is ignored.

It is possible for several grabs of different devices by the same or different clients to be active simultaneously. If a device is frozen on behalf of any grab, no event processing is performed for the device. It is possible for a single device to be frozen because of several grabs. In this case, the freeze must be released on behalf of each grab before events can again be processed.

XAllowDeviceEvents can generate a BadDevice or BadValue error.

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadValue

An invalid mode was specified on the request.

SEE ALSO

XGrabDevice()

Programming With Xlib

NAME

XGetFeedbackControl(), XChangeFeedbackControl() – query and change input device feedbacks

SYNOPSIS

```
XFeedbackState *XGetFeedbackControl (Display *display,
                                     XDevice *device,
                                     int *num_feedbacks)
int XChangeFeedbackControl (Display *display, XDevice *device,
                           Mask mask, XFeedbackControl *control )
```

ARGUMENTS

display
Specifies the connection to the X server.

device
Specifies the device whose feedbacks are to be queried or modified.

num_feedbacks
Specifies an address into which the number of feedbacks supported by the device is to be returned.

mask
Specifies a mask specific to each type of feedback that describes how the feedback is to be modified.

control
*Specifies the address of an **XFeedbackControl** structure that contains the new values for the feedback.*

DESCRIPTION

These requests are provided to manipulate those input devices that support feedbacks. A BadMatch error will be generated if the requested device does not support feedbacks. Whether or not a given device supports feedbacks can be determined by examining the information returned by the *XOpenDevice()* request. For those devices that support feedbacks, *XOpenDevice()* return **XInputClassInfo** structure with the *input_class* field equal to the constant FeedbackClass (defined in the file <X11/XI.h>).

The *XGetFeedbackControl()* request returns a pointer to a list of **XFeedbackState** structures. Each item in this list describes one of the feedbacks supported by the device. The items are variable length, so each contains its length to allow traversal to the next item in the list.

The feedback classes that are currently defined are: KbdFeedbackClass, PtrFeedbackClass, StringFeedbackClass, IntegerFeedbackClass, LedFeedbackClass, and BellFeedbackClass. These constants are defined in the file <X11/XI.h>. An input device may support zero or more classes of feedback, and may support multiple feedbacks of the same class. Each feedback contains a class identifier and an id that is unique within that class for that input device. The id is used to identify the feedback when making an *XChangeFeedbackControl()* request.

XGetFeedbackControl() can generate a BadDevice or BadMatch error.

The *XChangeFeedbackControl()* request modifies the values of one feedback on the specified device. The feedback is identified by the id field of the **XFeedbackControl**

structure that is passed with the request. The fields of the feedback that are to be modified are identified by the bits of the mask that is passed with the request.

XChangeFeedbackControl() can generate a BadDevice, BadMatch, or BadValue error.

STRUCTURES

Each class of feedback is described by a structure specific to that class. These structures are defined in the file `<X11/extensions/XInput.h>`. **XFeedbackState** and **XFeedbackControl** are generic structures that contain three fields that are at the beginning of each class of feedback:

```
typedef struct {
    XID class;
    int length;
    XID id;
} XFeedbackState, XFeedbackControl;
```

The

XKbdFeedbackState

structure defines the attributes that are returned for feedbacks equivalent to those on the X keyboard.

```
typedef struct {
    XID    class;
    int    length;
    XID    id;
    int    click;
    int    percent;
    int    pitch;
    int    duration;
    int    led_mask;
    int    global_auto_repeat;
    char    auto_repeats[32];
} XKbdFeedbackState;
```

The **XPtrFeedbackState** structure defines the attributes that are returned for feedbacks equivalent to those on the the X pointer.

```
typedef struct {
    XID    class;
    int    length;
    XID    id;
    int    accelNum;
    int    accelDenom;
    int    threshold;
} XPtrFeedbackState;
```

The **XIntegerFeedbackState** structure defines attributes that are returned for integer feedbacks.

```
typedef struct {
    XID    class;
    int    length;
    XID    id;
    int    resolution;
    int    minVal;
```

```

        int      maxVal;
    } XIntegerFeedbackState;

```

The **XStringFeedbackState** structure defines the attributes that are returned for string feedbacks.

```

typedef struct {
    XID      class;
    int      length;
    XID      id;
    int      max_symbols;
    int      num_syms_supported;
    KeySym   *syms_supported;
} XStringFeedbackState;

```

The **XBellFeedbackState** structure defines the attributes that are returned for bell feedbacks.

```

typedef struct {
    XID      class;
    int      length;
    XID      id;
    int      percent;
    int      pitch;
    int      duration;
} XBellFeedbackState;

```

The **XLedFeedbackState** structure defines the attributes that are returned for LED feedbacks.

```

typedef struct {
    XID      class;
    int      length;
    XID      id;
    int      led_values;
} XLedFeedbackState;

```

The **XPtrFeedbackControl** structure defines the attributes that can be controlled for pointer feedbacks.

```

typedef struct {
    XID      class;
    int      length;
    XID      id;
    int      accelNum;
    int      accelDenom;
    int      threshold;
} XPtrFeedbackControl;

```

The **XKbdFeedbackControl** structure defines the attributes that can be controlled for keyboard feedbacks.

```

typedef struct {
    XID      class;
    int      length;
    XID      id;

```

```

        int    click;
        int    percent;
        int    pitch;
        int    duration;
        int    led_mask;
        int    led_value;
        int    key;
        int    auto_repeat_mode;
    } XKbdFeedbackControl;

```

The **XStringFeedbackControl** structure defines the attributes that can be controlled for string feedbacks.

```

typedef struct {
    XID    class;
    int    length;
    XID    id;
    int    num_keysyms;
    KeySym *syms_to_display;
} XStringFeedbackControl;

```

The **XIntegerFeedbackControl** structure defines the attributes that can be controlled for integer feedbacks.

```

typedef struct {
    XID    class;
    int    length;
    XID    id;
    int    int_to_display;
} XIntegerFeedbackControl;

```

The **XBellFeedbackControl** structure defines the attributes that can be controlled for bell feedbacks.

```

typedef struct {
    XID    class;
    int    length;
    XID    id;
    int    percent;
    int    pitch;
    int    duration;
} XBellFeedbackControl;

```

The **XLedFeedbackControl** structure defines the attributes that can be controlled for LED feedbacks.

```

typedef struct {
    XID    class;
    int    length;
    XID    id;
    int    led_mask;
    int    led_values;
} XLedFeedbackControl;

```

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if some other client has caused the specified device to become the X keyboard or X pointer device via the *XChangeKeyboardDevice()* or *XChangePointerDevice()* requests.

BadMatch

This error may occur if an *XGetFeedbackControl()* request was made specifying a device that has no feedbacks, or an *XChangeFeedbackControl()* request was made with an **XFeedbackControl** structure that contains an invalid feedback type. It may also occur if an invalid combination of mask bits is specified (Key but no *AutoRepeatMode* for keyboard feedbacks), or if an invalid *KeySym* is specified for a string feedback.

BadValue

Some numeric value falls outside the range of values accepted by the *XChangeFeedbackControl()* request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

Programming With Xlib

NAME

XGetDeviceKeyMapping(), XChangeDeviceKeyMapping() – query or change device key mappings

SYNOPSIS

```
XChangeDeviceKeyMapping (Display *display, XDevice *device,
                        int first_keycode, int keysyms_per_keycode,
                        KeySym *keysyms, int keycode_count,
KeySym *XGetDeviceKeyMapping (Display *display, XDevice *device,
                        KeyCode first_keycode,
                        int keycode_count,
                        int *keysyms_per_keycode_return )
```

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device whose key mapping is to be queried or modified.
first_keycode
 Specifies the first KeyCode to be returned.
keycode_count
 Specifies the number of KeyCodes to be returned or modified.
keysyms_per_keycode
 Specifies the number of KeySyms per KeyCode.
keysyms_per_keycode_return
 Specifies the address of a variable into which the number of KeySyms per
 KeyCode will be returned.
keysyms Specifies the address of an array of KeySyms.

DESCRIPTION

For the specified device, the *XGetDeviceKeyMapping()* request returns the symbols for the specified number of KeyCodes starting with *first_keycode*. The value specified in *first_keycode* must be greater than or equal to *min_keycode* as returned by *XListInputDevices()*, or a BadValue error results. In addition, the following expression must be less than or equal to *max_keycode* as returned by *XListInputDevices()*:

$$first_keycode + keycode_count - 1$$

If this is not the case, a BadValue error results. The number of elements in the KeySyms list is:

$$keycode_count * keysyms_per_keycode_return$$

KeySym number N, counting from zero, for KeyCode K has the following index in the list, counting from zero: $(K - first_code) * keysyms_per_code_return + N$

The X server arbitrarily chooses the *keysyms_per_keycode_return* value to be large enough to report all requested symbols. A special KeySym value of NoSymbol is used to fill in unused elements for individual KeyCodes. To free the storage returned by *XGetDeviceKeyMapping()*, use *XFree()*.

If the specified device does not support input class keys, a BadMatch error will result.

XGetDeviceKeyMapping() can generate a BadDevice, BadMatch, or BadValue error.

For the specified device, the *XChangeDeviceKeyMapping()* request defines the symbols for the specified number of KeyCodes starting with *first_keycode*. The symbols for

KeyCode values outside this range remain unchanged. The number of elements in `keysyms` must be:

$$\text{num_codes} * \text{keysyms_per_keycode}$$

The specified `first_keycode` must be greater than or equal to `min_keycode` returned by `XListInputDevices()`, or a `BadValue` error results. In addition, the following expression must be less than or equal to `max_keycode` as returned by `XListInputDevices()`, or a `BadValue` error results:

$$\text{first_keycode} + \text{num_codes} - 1$$

KeySym number `N`, counting from zero, for KeyCode `K` has the following index in `keysyms`, counting from zero:

$$(\text{K} - \text{first_keycode}) * \text{keysyms_per_keycode} + \text{N}$$

The specified `keysyms_per_keycode` can be chosen arbitrarily by the client to be large enough to hold all desired symbols. A special KeySym value of `NoSymbol` should be used to fill in unused elements for individual KeyCode values. It is legal for `NoSymbol` to appear in nontrailing positions of the effective list for a KeyCode.

`XChangeDeviceKeyMapping()` generates a `DeviceMappingNotify` event that is sent to all clients that have selected that type of event.

There is no requirement that the X server interpret this mapping. It is merely stored for reading and writing by clients.

If the specified device does not support input class keys, a `BadMatch` error results.

`XChangeDeviceKeyMapping()` can generate a `BadDevice`, `BadMatch`, `BadAlloc`, or `BadValue` error.

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via `XOpenInputDevice`. This error may also occur if the specified device is the X keyboard or X pointer device.

BadMatch

This error may occur if an `XGetDeviceKeyMapping()` or `XChangeDeviceKeyMapping()` request was made specifying a device that has no keys.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadAlloc

The server failed to allocate the requested resource or server memory.

SEE ALSO

`XSetDeviceButtonMapping()` `XSetDeviceModifierMapping()` Programming With Xlib

NAME

XGetDeviceModifierMapping(), XSetDeviceModifierMapping() – query or change device modifier mappings

SYNOPSIS

```
XSetDeviceModifierMapping (Display *display, XDevice *device,
                           XModifierKeymap *modmap)
XModifierKeymap *XGetDeviceModifierMapping (Display *display,
                                             XDevice *device)
```

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device whose modifier mapping is to be queried or modified.
modmap Specifies a pointer to the **XModifierKeymap** structure.

DESCRIPTION

The *XSetDeviceModifierMapping()* request specifies the KeyCodes of the keys (if any) that are to be used as modifiers for the specified device. If it succeeds, the X server generates a DeviceMappingNotify event, and *XSetDeviceModifierMapping()* returns MappingSuccess. X permits at most eight modifier keys. If more than eight are specified in the **XModifierKeymap** structure, a BadLength error results.

The *modifiermap* member of the **XModifierKeymap** structure contains eight sets of *max_keypermod* KeyCodes, one for each modifier in the order Shift, Lock, Control, Mod1, Mod2, Mod3, Mod4, and Mod5. Only nonzero KeyCodes have meaning in each set, and zero KeyCodes are ignored. In addition, all of the nonzero KeyCodes must be in the range specified by *min_keycode* and *max_keycode* as returned by *XListInputDevices()*, or a BadValue error results. No KeyCode may appear twice in the entire map, or a BadValue error results.

An X server can impose restrictions on how modifiers can be changed, for example, if certain keys do not generate up transitions in hardware, if auto-repeat cannot be disabled on certain keys, or if multiple modifier keys are not supported. If some such restriction is violated, the status reply is MappingFailed, and none of the modifiers are changed. If the new KeyCodes specified for a modifier differ from those currently defined and any (current or new) keys for that modifier are in the logically down state, *XSetDeviceModifierMapping()* returns MappingBusy, and none of the modifiers is changed.

XSetDeviceModifierMapping() can generate BadLength, BadDevice, BadMatch, BadAlloc, and BadValue errors.

The *XGetDeviceModifierMapping()* request returns a pointer to a newly created **XModifierKeymap** structure that contains the keys being used as modifiers. The structure should be freed after use by calling *XFreeModifierMapping()*. If only zero values appear in the set for any modifier, that modifier is disabled.

XGetDeviceModifierMapping() can generate BadDevice and BadMatch errors.

STRUCTURES

The **XModifierKeymap** structure contains:

```
typedef struct {
    int max_keypermod;
```

```
        KeyCode *modifiermap;  
    } XModifierKeymap;
```

DIAGNOSTICS**BadLength**

More than eight keys were specified in the **XModifierKeymap** structure.

BadAlloc

The server failed to allocate the requested resource or server memory.

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadMatch

This error may occur if an *XGetDeviceModifierMapping()* or *XChangeDeviceModifierMapping()* request was made specifying a device that has no keys.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

XSetDeviceButtonMapping(), *XSetDeviceKeyMapping()*

Programming With Xlib

NAME

XChangeDeviceDontPropagateList(), XGetDeviceDontPropagateList() – query or change the dont-propagate-list for extension devices

SYNOPSIS

```
XChangeDeviceDontPropagateList (Display *display, Window *window,
                                int *count, XEventClass *event_list,
                                int mode)
XEventClass *XGetDeviceDontPropagateList (Display *display,
                                           Window *window,
                                           int *count)
```

ARGUMENTS

display Specifies the connection to the X server.
window Specifies the window whose dont-propagate-list is to be queried or modified.
event_list Specifies a pointer to a list of event classes.
mode Specifies the mode. You can pass *AddToList* , or *DeleteFromList*.
count Specifies the number of event classes in the list.

DESCRIPTION

The *XChangeDeviceDontPropagateList()* request modifies the list of events that should not be propagated to ancestors of the event window. This request allows extension events to be added to or deleted from that list. By default, all events are propagated to ancestor windows. Once modified, the list remains modified for the life of the window. Events are not removed from the list because the client that added them has terminated.

Suppression of event propagation is not allowed for all input extension events. If a specified event class is one that cannot be suppressed, a *BadClass* error will result. Events that can be suppressed include *DeviceKeyPress*, *DeviceKeyRelease*, *DeviceButtonPress*, *DeviceButtonRelease*, *DeviceMotionNotify*, *ProximityIn*, and *ProximityOut*.

XChangeDeviceDontPropagateList() can generate a *BadDevice*, *BadClass*, or *BadValue* error.

The *XGetDeviceDontPropagateList()* request queries the list of events that should not be propagated to ancestors of the event window.

XGetDeviceDontPropagateList() can generate a *BadClass* or *BadWindow* error.

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if some other client has caused the specified device to become the X keyboard or X pointer device via the *XChangeKeyboardDevice()* or *XChangePointerDevice()* requests.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

An invalid window id was specified.

BadClass

An invalid event class was specified.

SEE ALSO

Programming With Xlib

NAME

XChangeKeyboardDevice() – change which device is used as the X keyboard

SYNOPSIS

Status XChangeKeyboardDevice (Display **display*, XDevice **device*)

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device to be used as the X keyboard.

DESCRIPTION

The *XChangeKeyboardDevice()* request causes the server to use the specified device as the X keyboard. The device must have been previously opened by the requesting client via *XOpenDevice()* or a BadDevice error will result. The device must support input class Keys, or a BadMatch error will result. If the server implementation does not support using the requested device as the X keyboard, a BadDevice error will result.

If the specified device is grabbed by another client, AlreadyGrabbed is returned. If the specified device is frozen by a grab on another device, GrabFrozen is returned. If the request is successful, Success is returned.

If the request succeeds, a ChangeDeviceNotify event is sent to all clients that have selected that event. A MappingNotify event with request = MappingKeyboard is sent to all clients. The specified device becomes the X keyboard and the old X keyboard becomes accessible through the input extension protocol requests.

XChangeKeyboardDevice() can generate a BadDevice or a BadMatch error.

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist, has not been opened by this client via *XOpenInputDevice()*, or is already one of the core X device (pointer or keyboard). This error may also occur if the server implementation does not support using the specified device as the X keyboard.

BadMatch

This error may occur if an *XChangeKeyboardDevice()* request was made specifying a device that has no keys.

SEE ALSO

XChangePointerDevice()

Programming With Xlib

NAME

XChangePointerDevice() – change which device is the X pointer

SYNOPSIS

```
Status XChangePointerDevice (Display *display, XDevice *device,
                             int xaxis, int yaxis)
```

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device to be used as the X pointer.
xaxis Specifies the axis of the device to be used as the X pointer x-axis.
yaxis Specifies the axis of the device to be used as the X pointer y-axis.

DESCRIPTION

The *XChangePointerDevice()* request causes the server to use the specified device as the X pointer. The device must have been opened by the client via *XOpenDevice()* or a BadDevice error will result. The device must support input class Valuator or a BadMatch error will result. If the implementation does not support use of the specified device as the X pointer, a BadDevice error will result.

If the specified device is grabbed by another client, AlreadyGrabbed is returned. If the specified device is frozen by a grab on another device, GrabFrozen is returned. If the request is successful, Success is returned.

If the request succeeds, a ChangeDeviceNotify event is sent to all clients that have selected that event. A MappingNotify event with request = MappingPointer is sent to all clients. The specified device becomes the X pointer, and the old X pointer becomes accessible through the input extension protocol requests.

XChangePointerDevice() can generate a BadDevice or a BadMatch error.

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist, has not been opened by this client via *XOpenInputDevice()*, or is already one of the core X input devices (pointer or keyboard). This error may also occur if the server implementation does not support using the specified device as the X pointer.

BadMatch

This error may occur if an *XChangePointerDevice()* request was made specifying a device that has less than two valuator, or specifying a valuator index beyond the range supported by the device.

SEE ALSO

XChangeKeyboardDevice()

Programming With Xlib

NAME

XDeviceBell() – ring a bell on a device supported through the input extension

SYNOPSIS

```
Status XDeviceBell (Display *display, XDevice *device,
                    XID *feedbackclass, XID *feedbackid,
                    int *percent)
```

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device with which the bell is associated.
feedbackclass
 Specifies the class of the feedback with which the bell is associated.
feedbackid
 Specifies the id of the feedback with which the bell is associated.
percent Specifies the volume in the range -100 to 100 at which the bell should be rung.

DESCRIPTION

The *XDeviceBell()* request causes the server to ring a bell on the specified feedback of the specified device, if possible. The specified volume is relative to the base volume for the bell. If an invalid device is specified, a BadDevice error will be returned. The feedbackclass and feedbackid parameters contain values returned by an *XGetFeedbackControl()* request and uniquely identify the bell to ring. If a feedbackclass is specified that does not support a bell, or if a nonexistent feedbackid is specified, or a percent value is specified that is not in the range -100 to 100, a BadValue error will be returned.

The volume at which the bell is rung when the percent argument is nonnegative is:

$$\text{base} - [(\text{base} * \text{percent}) / 100] + \text{percent}$$

The volume at which the bell rings when the percent argument is negative is:

$$\text{base} + [(\text{base} * \text{percent}) / 100]$$

To change the base volume of the bell, use *XChangeFeedbackControl()*.

XDeviceBell() can generate a BadDevice or a BadValue error.

DIAGNOSTICS

BadDevice
 An invalid device was specified. The specified device does not exist, or has not been opened by this client via *XOpenInputDevice()*.
 BadValue
 An invalid feedbackclass, feedbackid, or percent value was specified.

SEE ALSO

XChangeFeedbackControl(), *XBell()*

Programming With Xlib

NAME

XGetDeviceMotionEvents(), XDeviceTimeCoord() – get device motion history

SYNOPSIS

```
XDeviceTimeCoord *XGetDeviceMotionEvents (Display *display,
                                           XDevice *device,
                                           Time start,
                                           Time stop,
                                           int *nevents_return,
                                           int *mode_return,
                                           int *axis_count_return)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device whose motion history is to be queried.

start

stop Specify the time interval in which the events are returned from the motion history buffer. You can pass a timestamp or CurrentTime.

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

mode_return
Returns the mode of the device (Absolute or Relative).

axis_count_return
Returns the count of axes being reported.

DESCRIPTION

The server may retain the recent history of the device motion and do so to a finer granularity than is reported by DeviceMotionNotify events. The XGetDeviceMotionEvents() request makes this history available.

The XGetDeviceMotionEvents() request returns all events in the motion history buffer that fall between the specified start and stop times, inclusive. If the start time is later than the stop time or if the start time is in the future, no events are returned. If the stop time is in the future, it is equivalent to specifying CurrentTime.

The *mode* indicates whether the device is reporting absolute positional data (mode=Absolute) or relative motion data (mode=Relative). Some devices allow their mode to be changed via the XSetDeviceMode() request. These constants are defined in the file <Xlib/XI.h>. The *axis_count* returns the number of axes or valuator being reported by the device.

XGetDeviceMotionEvents() can generate a BadDevice, or BadMatch error.

STRUCTURES

The **XDeviceTimeCoord** structure contains:

```
typedef struct {
    Time time;
    int *data;
} XDeviceTimeCoord;
```

The *time* member is set to the time, in milliseconds. The *data* member is a pointer to an array of integers. These integers are set to the values of each valuator or axis reported by the device. There is one element in the array per axis of motion reported by the device.

The value of the array elements depends on the mode of the device. If the mode is Absolute, the values are the raw values generated by the device. These may be scaled by client programs using the maximum values that the device can generate. The maximum value for each axis of the device is reported in the *max_val* field of the **XAxisInfo** returned by the *XListInputDevices()* request. If the mode is Relative, the data values are the relative values generated by the device.

You should use *XFreeDeviceMotionEvents()* to free the data returned by this request.

Errors returned by this request: BadDevice, BadMatch.

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadMatch

This error may occur if an *XGetDeviceMotionEvents()* request is made specifying a device that has no valuator and reports no axes of motion.

SEE ALSO

Programming With Xlib

NAME

XGrabDeviceButton(), XUngrabDeviceButton() – grab / ungrab extension input device buttons

SYNOPSIS

```
XGrabDeviceButton (Display *display, XDevice *device,
                  unsigned int button, unsigned int modifiers,
                  XDevice *modifier_device, Window grab_window,
                  Bool owner_events, unsigned int event_count,
                  XEventClass *event_list, int this_device_mode,
                  int other_devices_mode)
XUngrabDeviceButton (Display *display, XDevice *device,
                   unsigned int button, unsigned int modifiers,
                   XDevice *modifier_device, Window grab_window)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device that is to be grabbed or released

button Specifies the device button that is to be grabbed or released or AnyButton.

modifiers Specifies the set of keymasks or AnyModifier. The mask is the bitwise inclusive OR of the valid keymask bits. Valid bits are: Shiftmask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, Mod5Mask.

modifier_device specifies the device whose modifiers are to be used. If the modifier_device specified is NULL, the X keyboard will be used as the modifier_device.

grab_window Specifies the grab window.

owner_events Specifies a Boolean value that indicates whether the device events are to be reported as usual or reported with respect to the grab window if selected by the event list.

event_count Specifies the number of event classes in the event list.

event_list Specifies which events are reported to the client.

this_device_mode Specifies further processing of events from this device. You can pass GrabModeSync or GrabModeAsync.

other_devices_mode Specifies further processing of events from all other devices. You can pass GrabModeSync or GrabModeAsync.

DESCRIPTION

The *XGrabDeviceButton()* request establishes a passive grab. In the future, the device is actively grabbed (as for *XGrabDevice()*), the last-grab time is set to the time at which the button was pressed (as transmitted in the DeviceButtonPress event), and the DeviceButtonPress event is reported if all of the following conditions are true:

- The device is not grabbed, and the specified button is logically pressed when the specified modifier keys are logically down on the specified modifier device and no

other buttons or modifier keys are logically down.

- Either the grab window is an ancestor of (or is) the focus window, OR the grab window is a descendent of the focus window and contains the device.
- A passive grab on the same button / modifier combination does not exist on any ancestor of *grab_window*.

The interpretation of the remaining arguments is as for *XGrabDevice()*. The active grab is terminated automatically when the logical state of the device has all buttons released (independent of the logical state of the modifier keys).

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

This request overrides all previous grabs by the same client on the same button / modifier combinations on the same window. A modifiers of AnyModifier is equivalent to issuing the grab request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned KeyCodes. A button of AnyButton is equivalent to issuing the request for all possible buttons. Otherwise, it is not required that the specified button currently be assigned to a physical button.

A *modifier_device* of NULL indicates that the X keyboard is to be used as the *modifier_device*.

If some other client has already issued a *XGrabDeviceButton()* with the same button / modifier combination on the same window, a BadAccess error results. When using AnyModifier or AnyButton, the request fails completely, and a BadAccess error results (no grabs are established) if there is a conflicting grab for any combination. *XGrabDeviceButton* has no effect on an active grab.

XGrabDeviceButton can generate BadClass, BadDevice, BadMatch, BadValue, and BadWindow errors.

The *XUngrabDeviceButton* request releases the passive grab for a button / modifier combination on the specified window if it was grabbed by this client. A modifier of AnyModifier is equivalent to issuing the ungrab request for all possible modifier combinations, including the combination of no modifiers. A button of AnyButton is equivalent to issuing the request for all possible buttons. *XUngrabDeviceButton()* has no effect on an active grab.

A *modifier_device* of NULL indicates that the X keyboard should be used as the *modifier_device*.

XUngrabDeviceButton() can generate BadDevice, BadMatch, BadValue and BadWindow errors.

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadMatch

This error may occur if an *XGrabDeviceButton()* request was made specifying a device that has no buttons, or specifying a modifier device that has no keys.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowDeviceEvents(),

XGrabDevice(),

XGrabDeviceKey(),

Programming With Xlib

NAME

XGrabDeviceKey(), XUngrabDeviceKey() – grab / ungrab extension input device Keys

SYNOPSIS

```
XGrabDeviceKey (Display *display, XDevice *device,
               unsigned int Key, unsigned int modifiers,
               XDevice *modifier_device, Window grab_window,
               Bool owner_events, unsigned int event_count,
               XEventClass event_list, int this_device_mode,
               int other_devices_mode)
XUngrabDeviceKey (Display *display, XDevice *device,
                 unsigned int Key, unsigned int modifiers,
                 XDevice *modifier_device, Window grab_window)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device that is to be grabbed or released

Key Specifies the device Key that is to be grabbed or released or *AnyKey*.

modifiers Specifies the set of keymasks or *AnyModifier*. The mask is the bitwise inclusive OR of the valid keymask bits. Valid bits are: ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, Mod5Mask.

modifier_device Specifies the device whose modifiers are to be used. If a *modifier_device* of NULL is specified, the X keyboard will be used as the *modifier_device*.

grab_window Specifies the grab window.

owner_events Specifies a Boolean value that indicates whether the device events are to be reported as usual or reported with respect to the grab window if selected by the event list.

event_count Specifies the number of event classes in the event list.

event_list Specifies which device events are reported to the client.

this_device_mode Specifies further processing of events from this device. You can pass GrabModeSync or GrabModeAsync.

other_devices_mode Specifies further processing of events from other devices. You can pass GrabModeSync or GrabModeAsync.

DESCRIPTION

The *XGrabDeviceKey()* request establishes a passive grab. In the future, the device is actively grabbed (as for *XGrabDevice()*), the last-device-grab time is set to the time at which the *Key* was pressed (as transmitted in the DeviceKeyPress event), and the DeviceKeyPress event is reported if all of the following conditions are true:

- The device is not grabbed, and the specified key is logically pressed when the specified modifier keys are logically down, and no other keys or modifier keys are logically down.

- The grab_window is an ancestor (or is) the focus window OR the grab window is a descendant of the focus window and contains the device.
- The confine_to window (if any) is viewable.
- A passive grab on the same key / modifier combination does not exist on any ancestor of grab_window.

The interpretation of the remaining arguments is as for *XGrabDevice()*. The active grab is terminated automatically when the logical state of the device has the specified key released.

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

If the key is not AnyKey, it must be in the range specified by *min_keycode* and *max_keycode* as returned by the *XListInputDevices()* request. Otherwise, a BadValue error results.

This request overrides all previous grabs by the same client on the same Key / modifier combinations on the same window. A modifier of AnyModifier is equivalent to issuing the grab request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned KeyCodes. A key of AnyKey is equivalent to issuing the request for all possible keys. Otherwise, it is not required that the specified key currently be assigned to a physical Key.

If a *modifier_device* of NULL is specified, the X keyboard will be used as the *modifier_device*.

If some other client has already issued a *XGrabDeviceKey()* with the same Key / modifier combination on the same window, a BadAccess error results. When using AnyModifier or AnyKey , the request fails completely, and a BadAccess error results (no grabs are established) if there is a conflicting grab for any combination. *XGrabDeviceKey()* has no effect on an active grab.

XGrabDeviceKey() can generate BadAccess, BadClass, BadDevice, BadMatch, BadValue, and BadWindow errors. It returns Success on successful completion of the request.

The *XUngrabDeviceKey()* request releases the passive grab for a key / modifier combination on the specified window if it was grabbed by this client. A modifier of AnyModifier is equivalent to issuing the ungrab request for all possible modifier combinations, including the combination of no modifiers. A Key of AnyKey is equivalent to issuing the request for all possible Keys. *XUngrabDeviceKey()* has no effect on an active grab.

If a *modifier_device* of NULL is specified, the X keyboard will be used as the *modifier_device*.

XUngrabDeviceKey() can generate BadDevice, BadMatch, BadValue and BadWindow errors.

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadMatch

This error may occur if an *XGrabDeviceKey()* request was made specifying a device that has no keys, or a modifier device that has no keys.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowDeviceEvents(), *XGrabDevice()*, *XGrabDeviceButton()*

Programming With Xlib

NAME

XGrabDevice(), XUngrabDevice() – grab / release the specified extension device

SYNOPSIS

```
int XGrabDevice (Display *display, XDevice *device,
                Window grab_window, Bool owner_events,
                int event_count, XEventClass *event_list,
                int this_device_mode, int other_devices_mode,
                Time time)
XUngrabDevice (Display *display, XDevice *device, Time time)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device to be grabbed or released.

grab_window
Specifies the id of a window to be associated with the device.

owner_events
Specifies a Boolean value that indicates whether the events from the device are to be reported as usual or reported with respect to the grab window if selected by the event list.

event_count
Specifies the number of elements in the *event_list* array.

event_list
Specifies a pointer to a list of event classes that indicates which events the client wishes to receive. These event classes must have been obtained specifying the device being grabbed.

this_device_mode
Specifies further processing of events from this device. You can pass GrabModeSync or GrabModeAsync.

other_devices_mode
Specifies further processing of events from other devices. You can pass GrabModeSync or GrabModeAsync.

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

DESCRIPTION

The *XGrabDevice()* request actively grabs control of the device and generates DeviceFocusIn and DeviceFocusOut events. Further device events are reported only to the grabbing client. *XGrabDevice()* overrides any active device grab by this client. *event_list* is a pointer to a list of event classes. This list indicates which events the client wishes to receive while the grab is active. If *owner_events* is False, all generated device events are reported with respect to *grab_window* if selected. If *owner_events* is True and if a generated device event would normally be reported to this client, it is reported normally; otherwise, the event is reported with respect to the *grab_window*, and is only reported if specified in the *event_list*.

If the *this_device_mode* argument is GrabModeAsync, device event processing continues as usual. If the device is currently frozen by this client, then processing of device events is resumed. If the *this_device_mode* argument is GrabModeSync, the state of the device (as seen by client applications) appears to freeze, and the X server generates no further device events until the grabbing client issues a releasing *XAllowDeviceEvents()* call or until the device grab is released.

Actual device changes are not lost while the device is frozen; they are simply queued in the server for later processing.

If *other_devices_mode* is *GrabModeAsync*, processing of events from other devices is unaffected by activation of the grab. If *other_devices_mode* is *GrabModeSync*, the state of all devices except the grabbed device (as seen by client applications) appears to freeze, and the X server generates no further events from those devices until the grabbing client issues a releasing *XAllowDeviceEvents()* call or until the device grab is released. Actual events are not lost while the devices are frozen; they are simply queued in the server for later processing.

If the device is actively grabbed by some other client, *XGrabDevice()* fails and returns *AlreadyGrabbed*. If *grab_window()* is not viewable, it fails and returns *GrabNotViewable*. If the device is frozen by an active grab of another client, it fails and returns *GrabFrozen*. If the specified time is earlier than the last-device-grab time or later than the current X server time, it fails and returns *GrabInvalidTime*. Otherwise, the last-device-grab time is set to the specified time (*CurrentTime* is replaced by the current X server time).

If a grabbed device is closed by a client while an active grab by that client is in effect, the active grab is released. If the device is frozen only by an active grab of the requesting client, it is thawed.

XGrabDevice() can generate *BadClass*, *BadDevice*, *BadValue*, and *BadWindow* errors.

The *XUngrabDevice()* request releases the device and any queued events if this client has it actively grabbed from either *XGrabDevice()* or *XGrabDeviceKey()*. If other devices are frozen by the grab, *XUngrabDevice()* thaws them. *XUngrabDevice()* does not release the device and any queued events if the specified time is earlier than the last-device-grab time or is later than the current X server time. It also generates *DeviceFocusIn* and *DeviceFocusOut* events. The X server automatically performs an *UngrabDevice()* request if the event window for an active device grab becomes not viewable.

XUngrabDevice() can generate a *BadDevice* error.

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XAllowDeviceEvents(), *XGrabDeviceButton()*, *XGrabDeviceKey()*,

Programming With Xlib

NAME

XListInputDevices(), XFreeDeviceList() – list available input devices

SYNOPSIS

```
XDeviceInfo *XListInputDevices (Display *display, int *ndevices_return)
XFreeDeviceList (XDeviceInfo *list)
```

ARGUMENTS

display Specifies the connection to the X server.

ndevices_return
Specifies a pointer to a variable where the number of available devices can be returned.

list Specifies the list of devices to free. The *XFreeDeviceList()* function frees the list of available extension input devices.

DESCRIPTION

The *XListInputDevices()* request lists the available extension input devices. This list includes the X pointer and X keyboard, any other input devices that are currently accessible through the X server, and any input devices that are not currently accessible through the X server but could be accessed if requested.

Some server implementations may make all input devices available at the time the server is initialized. Others may wait until requested by a client to access an input device. In the latter case, it is possible that an input device will be listed as available at one time but not at another.

For each input device available to the server, the *XListInputDevices* request returns an *XDeviceInfo* structure. That structure contains a pointer to a list of structures, each of which contains information about one class of input supported by the device.

The **XDeviceInfo** structure is defined as follows:

```
typedef struct _XDeviceInfo
{
    XID          id;
    Atom         type;
    char         *name;
    int          num_classes;
    int          use;
    XAnyClassPtr inputclassinfo;
} XDeviceInfo;
```

The *id* is a number in the range 0-128 that uniquely identifies the device. It is assigned to the device when it is initialized by the server.

The *type* field is of type *Atom* and indicates the nature of the device.

The *name* field contains a pointer to a null-terminated string that corresponds to one of the defined device types. The *name* will correspond to one of the following strings (defined in the header file **<X11/XL.h>**):

```
XI_MOUSE
XI_TABLET
```

```

XI_KEYBOARD
XI_TOUCHSCREEN
XI_TOUCHPAD
XI_BUTTONBOX
XI_BARCODE
XI_TRACKBALL
XI_QUADRATURE
XI_ID_MODULE
XI_ONE_KNOB
XI_NINE_KNOB
XI_KNOB_BOX
XI_SPACEBALL
XI_DATAGLOVE
XI_EYETRACKER
XI_CURSORKEYS
XI_FOOTMOUSE

```

These names may be directly compared with the *name* field of the **XDeviceInfo** structure, or used in an *XInternAtom()* request to return an atom that can be compared with the type field of the **XDeviceInfo** structure.

The *num_classes* field is a number in the range 0-255 that specifies the number of input classes supported by the device for which information is returned by *ListInputDevices*. Some input classes, such as class Focus and class Proximity do not have any information to be returned by *ListInputDevices()*.

The *use* field specifies how the device is currently being used. If the value is *IsXKeyboard*, the device is currently being used as the X keyboard. If the value is *IsXPointer*, the device is currently being used as the X pointer. If the value is *IsXExtensionDevice*, the device is available for use as an extension device.

The *inputclassinfo* field contains a pointer to the first input-class specific data. The first two fields are common to all classes.

The *class* field is a number in the range 0-255. It uniquely identifies the class of input for which information is returned. Currently defined classes are *KeyClass*, *ButtonClass*, and *ValuatorClass*.

The *length* field is a number in the range 0- 255. It specifies the number of bytes of data that are contained in this input class. The *length* includes the class and length fields.

The **XKeyInfo** structure describes the characteristics of the keys on the device. It is defined as follows:

```

typedef struct _XKeyInfo {
    XID          class;
    int          length;
    unsigned short min_keycode;
    unsigned short max_keycode;
    unsigned short num_keys;
} XKeyInfo;

```

min_keycode is of type KEYCODE. It specifies the minimum keycode that the device will report. The minimum keycode will not be smaller than 8.

max_keycode is of type KEYCODE. It specifies the maximum keycode that the device will report. The maximum keycode will not be larger than 255.

num_keys specifies the number of keys that the device has.

The **XButtonInfo** structure defines the characteristics of the buttons on the device. It is defined as follows:

```
typedef struct _XButtonInfo {
    XID          class;
    int          length;
    short        num_buttons;
} XButtonInfo;
```

num_buttons specifies the number of buttons that the device has.

The **XValuatorInfo** structure defines the characteristics of the valuator on the device. It is defined as follows:

```
typedef struct      _XValuatorInfo {
    XID              class;
    int              length;
    unsigned char    num_axes;
    unsigned char    mode;
    unsigned long    motion_buffer;
    XAxisInfoPtr     axes;
} XValuatorInfo;
```

num_axes contains the number of axes the device supports.

mode is a constant that has one of the following values: Absolute or Relative. Some devices allow the mode to be changed dynamically via the *SetDeviceMode()* request.

motion_buffer_size is a cardinal number that specifies the number of elements that can be contained in the motion history buffer for the device.

The *axes* field contains a pointer to an **XAxisInfo** structure.

The **XAxisInfo** structure is defined as follows:

```
typedef struct _XAxisInfo {
    int    resolution;
    int    min_value;
    int    max_value;
} XAxisInfo;
```

The *resolution* contains a number in counts / meter.

The *min_val* field contains a number that specifies the minimum value the device reports for this axis. For devices whose mode is Relative, the *min_val* field will contain 0.

The *max_val* field contains a number that specifies the maximum value the device reports for this axis. For devices whose mode is Relative, the *max_val* field will contain 0.

To free the **XDeviceInfo** array created by *XListInputDevices()*, use *XFreeDeviceList()*.

DIAGNOSTICS

None.

SEE ALSO

Programming With Xlib

NAME

XOpenDevice(), XCloseDevice() – open or close an extension input device

SYNOPSIS

```
XDevice *XOpenDevice (Display *display, XID device_id)
XCloseDevice (Display *display, XDevice *device)
```

ARGUMENTS

display Specifies the connection to the X server.
device_id Specifies the id of the device to be opened
device Specifies the device to be closed

DESCRIPTION

The *XOpenDevice()* request makes an input device accessible to a client through input extension protocol requests. If successful, it returns a pointer to an XDevice structure.

The *XCloseDevice()* request makes an input device inaccessible to a client through input extension protocol requests. Before terminating, and client that has opened input devices through the input extension should close them via *CloseDevice()*.

When a client makes an *XCloseDevice()* request, any active grabs that the client has on the device are released. Any event selections that the client has are deleted, as well as any passive grabs. If the requesting client is the last client accessing the device, the server may disable all access by X to the device.

XOpenDevice() and *XCloseDevice()* can generate a BadDevice error.

STRUCTURES

The **XDevice** structure returned by *XOpenDevice()* contains:

```
typedef struct {
    XID device_id;
    int num_classes;
    XInputClassInfo *classes;
} XDevice;
```

The *classes* field is a pointer to an array of **XInputClassInfo** structures. Each element of this array contains an event type base for a class of input supported by the specified device. The *num_classes* field indicates the number of elements in the classes array.

The **XInputClassInfo** structure contains:

```
typedef struct {
    unsigned char input_class;
    unsigned char event_type_base;
} XInputClassInfo;
```

The *input_class* field identifies one class of input supported by the device. Defined types include KeyClass, ButtonClass, ValuatorClass, ProximityClass, FeedbackClass, FocusClass, and OtherClass. The *event_type_base* identifies the event type of the first event in that class.

The information contained in the **XInputClassInfo** structure is used by macros to obtain the event classes that clients use in making *XSelectExtensionEvent()* requests. Currently defined macros include *DeviceKeyPress()*, *DeviceKeyRelease()*, *DeviceButtonPress()*,

DeviceButtonRelease(), *DeviceMotionNotify()*, *DeviceFocusIn()*, *DeviceFocusOut()*, *ProximityIn()*, *ProximityOut()*, *DeviceStateNotify()*, *DeviceMappingNotify()*, *ChangeDeviceNotify()*, *DevicePointerMotionHint()*, *DeviceButton1Motion()*, *DeviceButton2Motion()*, *DeviceButton3Motion()*, *DeviceButton4Motion()*, *DeviceButton5Motion()*, *DeviceButtonMotion()*, *DeviceOwnerGrabButton()*, *DeviceButtonPressGrab()*, and *NoExtensionEvent()*.

To obtain the proper event class for a particular device, one of the above macros is invoked using the **XDevice** structure for that device. For example,

```
DeviceKeyPress (*device, type, eventclass);
```

returns the *DeviceKeyPress* event type and the eventclass for *DeviceKeyPress* events from the specified device.

This eventclass can then be used in an *XSelectExtensionEvent()* request to ask the server to send *DeviceKeyPress* events from this device. When a selected event is received via *XNextEvent()*, the type can be used for comparison with the type in the event.

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist, or is the X keyboard or X pointer. This error may also occur if some other client has caused the specified device to become the X keyboard or X pointer device via the *XChangeKeyboardDevice()* or *XChangePointerDevice()* requests.

SEE ALSO

Programming With Xlib

NAME

XQueryDeviceState() – query the state of an extension input device.

SYNOPSIS

```
XQueryDeviceState (Display *display, XDevice *device)
```

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device whose state is to be queried.

DESCRIPTION

The *XQueryDeviceState()* request queries the state of an input device. The current state of keys and buttons (up or down), and valuator (current value) on the device is reported by this request. Each key or button is represented by a bit in the **XDeviceState** structure that is returned. Valuator on the device report 0 if they are reporting relative information, and the current value if they are reporting absolute information.

XQueryDeviceState() can generate a BadDevice error.

STRUCTURES

The structure contains:

```
typedef struct {
    XID device_id;
    int num_classes;
    XInputClass *data;
} XDeviceState;
```

The **XValuatorState** structure contains:

```
typedef struct {
    unsigned char class;
    unsigned char length;
    unsigned char num_valuators;
    unsigned char mode;
    int *valuators;
} XValuatorState;
```

The **XKeyState** structure contains:

```
typedef struct {
    unsigned char class;
    unsigned char length;
    short num_keys;
    char keys[32];
} XKeyState;
```

The **XButtonState** structure contains:

```
typedef struct {
    unsigned char class;
    unsigned char length;
    short num_buttons;
    char buttons[32];
} XButtonState;
```

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if some other client has caused the specified device to become the X keyboard or X pointer device via the *XChangeKeyboardDevice()* or *XChangePointerDevice()* requests.

SEE ALSO

Programming With Xlib

NAME

XSendExtensionEvent() – send input extension events to a client

SYNOPSIS

```
Status XSendExtensionEvent (Display *display, XDevice *device,
                             Window destination, Bool propagate,
                             int event_count, XEventClass *event_list,
                             XEvent *event_send)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device from which the events are to be sent.

destination Specifies the window the event is to be sent to. You can pass window id, PointerWindow, or InputFocus.

propagate Specifies a Boolean value that is either True or False.

event_count Specifies the count of XEventClasses in *event_list*.

event_list Specifies the list of event selections to be used.

event_send Specifies a pointer to the event that is to be sent.

DESCRIPTION

The *XSendExtensionEvent()* request identifies the destination window, determines which clients should receive the specified events, and ignores any active grabs. This request requires you to pass an event class list. For a discussion of the valid event class names, see *XOpenDevice()*. This request uses the *w* argument to identify the destination window as follows:

- If *w* is PointerWindow, the destination window is the window that contains the pointer.
- If *w* is InputFocus and if the focus window contains the pointer, the destination window is the window that contains the pointer; otherwise, the destination window is the focus window.

To determine which clients should receive the specified events, *XSendExtensionEvent()* uses the *propagate* argument as follows:

- If *event_list* is the empty set, the event is sent to the client that created the destination window. If that client no longer exists, no event is sent.
- If *propagate* is False, the event is sent to every client selecting on destination any of the event types specified by the *event_list* array.
- If *propagate* is True and no clients have selected on destination any of the events specified by the *event_list* array, the destination is replaced with the closest ancestor of destination for which some client has selected a type specified by the event-list array and for which no intervening window has that type in its do-not-propagate-mask. If no such window exists or if the window is an ancestor of the focus window and InputFocus was originally specified as the destination, the event is not sent to any clients. Otherwise, the event is reported to every client selecting on the final

destination any of the events specified in the event_list array.

The event in the **XEvent** structure must be one of the events defined by the input extension (or a BadValue error results) so that the X server can correctly byte-swap the contents as necessary. The contents of the event are otherwise unaltered and unchecked by the X server except to force *send_event* to True in the forwarded event and to set the serial number in the event correctly.

XSendExtensionEvent() returns zero if the conversion to wire protocol format failed and returns nonzero otherwise. *XSendExtensionEvent()* can generate BadClass, BadDevice, BadValue, and BadWindow errors.

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

SEE ALSO

XOpenDevice()

Programming With Xlib

NAME

XSetDeviceButtonMapping(), XGetDeviceButtonMapping() – query or change device button mappings

SYNOPSIS

```
int XSetDeviceButtonMapping (Display *display, XDevice *device,
                             unsigned char map[], int nmap)
int XGetDeviceButtonMapping (Display *display, XDevice *device,
                             unsigned char map_return[], int nmap)
```

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device whose button mapping is to be queried or changed.
map Specifies the mapping list.
map_return Returns the mapping list.
nmap Specifies the number of items in the mapping list.

DESCRIPTION

The *XSetDeviceButtonMapping()* request sets the mapping of the specified device. If it succeeds, the X server generates a DeviceMappingNotify event, and *XSetDeviceButtonMapping()* returns MappingSuccess. Element *map[i]* defines the logical button number for the physical button *i+1*. The length of the list must be the same as *XGetDeviceButtonMapping()* would return, or a BadValue error results. A zero element disables a button, and elements are not restricted in value by the number of physical buttons. However, no two elements can have the same nonzero value, or a BadValue error results. If any of the buttons to be altered are logically in the down state, *XSetDeviceButtonMapping()* returns MappingBusy, and the mapping is not changed.

XSetDeviceButtonMapping() can generate BadDevice, BadMatch, and BadValue errors.

The *XGetDeviceButtonMapping()* request returns the current mapping of the specified device. Buttons are numbered starting from one. *XGetDeviceButtonMapping()* returns the number of physical buttons actually on the device. The nominal mapping for a device is *map[i]=i+1*. The *nmap* argument specifies the length of the array where the device mapping is returned, and only the first *nmap* elements are returned in *map_return*.

XGetDeviceButtonMapping() can generate BadDevice or BadMatch errors.

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if the specified device is the X keyboard or X pointer device.

BadMatch

This error may occur if an *XGetDeviceButtonMapping()* or *XSetDeviceButtonMapping()* request was made specifying a device that has no buttons.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by

the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

SEE ALSO

*XChangeDeviceKeyboardControl, XChangeKeyboardControl(),
XChangeDeviceKeyMapping, XChangeDeviceModifierMapping
Programming With Xlib*

NAME

XSetDeviceFocus(), XGetDeviceFocus() – control extension input device focus

SYNOPSIS

```
XSetDeviceFocus (Display *display, Display *device,
                Window focus, int revert_to, Time time)
XGetDeviceFocus (Display *display, Display *device,
                Window *focus_return, int *revert_to_return,
                int *time_return)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device whose focus is to be queried or changed.

focus Specifies the window, PointerRoot, FollowKeyboard, or None.

focus_return
Returns the focus window, PointerRoot, FollowKeyboard, or None.

revert_to
Specifies where the input focus reverts to if the window becomes not viewable. You can pass RevertToParent, RevertToPointerRoot, RevertToFollowKeyboard, or RevertToNone.

revert_to_return
Returns the current focus state RevertToParent, RevertToPointerRoot, RevertToFollowKeyboard, or RevertToNone.

time_return
Returns the *last_focus_time* for the device.

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

DESCRIPTION

The XSetDeviceFocus request changes the focus of the specified device and its 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). *XSetDeviceFocus()* causes the X server to generate DeviceFocusIn and DeviceFocusOut events.

Depending on the *focus* argument, the following occurs:

- If *focus* is None, all device events are discarded until a new focus window is set, and the *revert_to* argument is ignored.
- If *focus* is a window, it becomes the device's focus window. If a generated device event would normally be reported to this window or one of its inferiors, the event is reported 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 event from the specified device. In this case, the *revert_to* argument is ignored.
- If *focus* is FollowKeyboard, the focus window is dynamically taken to be the window to which the X keyboard focus is set at each input event.

The specified focus window must be viewable at the time *XSetDeviceFocus()* is called, or a BadMatch error results. If the focus window later becomes not viewable, the X server evaluates the *revert_to* argument to determine the new focus window as follows:

- If *revert_to* is `RevertToParent`, the focus reverts to the parent (or the closest viewable ancestor), and the new *revert_to* value is taken to be `RevertToNone`.
- If *revert_to* is `RevertToPointerRoot`, `RevertToFollowKeyboard`, or `RevertToNone`, the focus reverts to `PointerRoot`, `FollowKeyboard`, or `None`, respectively.

When the focus reverts, the X server generates `DeviceFocusIn` and `DeviceFocusOut` events, but the last-focus-change time is not affected.

Input extension devices are not required to support the ability to be focused. Attempting to set the focus of a device that does not support this request will result in a `BadMatch` error. Whether or not given device can support this request can be determined by the information returned by `XOpenDevice()`. For those devices that support focus, `XOpenDevice()` will return an **XInputClassInfo** structure with the *input_class* field equal to the constant `FocusClass` (defined in the file `<X11/XI.h>`).

`XSetDeviceFocus()` can generate `BadDevice`, `BadMatch`, `BadValue`, and `BadWindow` errors.

The `XGetDeviceFocus()` request returns the focus window and the current focus state.

Not all input extension devices can be focused. Attempting to query the focus state of a device that can't be focused results in a `BadMatch` error. A device that can be focused returns information for input Class Focus when an `XOpenDevice()` request is made.

`XGetDeviceFocus()` can generate `BadDevice`, and `BadMatch` errors.

DIAGNOSTICS

BadDevice

An invalid device was specified. The specified device does not exist or has not been opened by this client via `XOpenInputDevice()`. This error may also occur if the specified device is the X keyboard or X pointer device.

BadValue

Some numeric value falls outside the range of values accepted by the request. Unless a specific range is specified for an argument, the full range defined by the argument's type is accepted. Any argument defined as a set of alternatives can generate this error.

BadWindow

A value for a Window argument does not name a defined Window.

BadMatch

This error may occur if an `XGetDeviceFocus()` or `XSetDeviceFocus()` request was made specifying a device that the server implementation does not allow to be focused.

SEE ALSO

Programming With Xlib

NAME

XSelectExtensionEvent(), XGetSelectedExtensionEvents() – select extension events, get the list of currently selected extension events

SYNOPSIS

```
XSelectExtensionEvent (Display *display, Window w,
                      XEventClass *event_list, int event_count)
XGetSelectedExtensionEvents (Display *display, Window w,
                           int this_client_event_count_return,
                           XEventClass *this_client_event_list_return,
                           int all_clients_event_count_return,
                           XEventClass *all_clients_event_list_return)
```

ARGUMENTS

display Specifies the connection to the X server.

w Specifies the window whose events you are interested in.

event_list Specifies the list of event classes that describe the events you are interested in.

event_count Specifies the count of event classes in the event list.

this_client_event_count_return Returns the count of event classes selected by this client.

this_client_event_list_return Returns a pointer to the list of event classes selected by this client.

all_clients_event_count_return Returns the count of event classes selected by all clients.

all_clients_event_list_return Returns a pointer to the list of event classes selected by all clients.

DESCRIPTION

The *XSelectExtensionEvent()* request causes the X server to report the events associated with the specified list of event classes. Initially, X will not report any of these events. Events are reported relative to a window. If a window is not interested in a device event, it usually propagates to the closest ancestor that is interested, unless the *do_not_propagate* mask prohibits it.

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 a DeviceButtonPress event with automatic passive grabbing enabled, which is associated with the event class DeviceButtonPressGrab. To receive DeviceButtonPress events without automatic passive grabbing, use event class DeviceButtonPress but do not specify event class DeviceButtonPressGrab.

The server reports the event to all interested clients.

Information contained in the **XDevice** structure returned by *XOpenDevice()* is used by macros to obtain the event classes that clients use in making *XSelectExtensionEvent()* requests. *Currently()* defined *DeviceKeyPress()*, *DeviceKeyRelease()*, *DeviceButtonPress()*,

DeviceButtonRelease(), *DeviceMotionNotify()*, *DeviceFocusIn()*, *DeviceFocusOut()*, *ProximityIn()*, *ProximityOut()*, *DeviceStateNotify()*, *DeviceMappingNotify()*, *ChangeDeviceNotify()*, *DevicePointerMotionHint()*, *DeviceButton1Motion()*, *DeviceButton2Motion()*, *DeviceButton3Motion()*, *DeviceButton4Motion()*, *DeviceButton5Motion()*, *DeviceButtonMotion()*, *DeviceOwnerGrabButton()*, *DeviceButtonPressGrab()*, and *() NoExtensionEvent()*.

To obtain the proper event class for a particular device, one of the above macros is invoked using the **XDevice** structure for that device. For example,

```
DeviceKeyPress (*device, type, eventclass);
```

returns the *DeviceKeyPress* event type and the eventclass for selecting *DeviceKeyPress* events from this device.

XSelectExtensionEvent() can generate a *BadWindow* or *BadClass* error. The *XGetSelectedExtensionEvents()* request reports the extension events selected by this client and all clients for the specified window. This request returns pointers to two **XEventClass** arrays. One lists the input extension events selected by this client from the specified window. The other lists the event classes selected by all clients from the specified window. You should use *XFree()* to free these two arrays.

XGetSelectedExtensionEvents() can generate a *BadWindow* error.

DIAGNOSTICS

BadWindow

A value for a *Window* argument does not name a defined *Window*.

BadClass

A value for an *XEventClass* argument is invalid.

SEE ALSO

Programming With Xlib

NAME

XSetDeviceValuators() – initialize the valuators on an extension input device

SYNOPSIS

```
XSetDeviceValuators (Display *display, XDevice *device,  
                     int *valuators, int first_valuator,  
                     int num_valuators)
```

ARGUMENTS

display Specifies the connection to the X server.

device Specifies the device whose valuators are to be initialized.

valuators
Specifies a pointer to an array of integer values to be used to initialize the device valuators.

first_valuator
Specifies the first valuator to be set. Valuators are numbered beginning with zero.

num_valuators
Specifies the number of valuators to be set.

DESCRIPTION

The *XSetDeviceValuators()* request sets the current values of the valuators of an input device. The valuators in the range *first_valuator* to (*first_valuator* + *num_valuators*) are set to the specified values. Valuators are numbered beginning with 0. Not all input devices support initialization of valuator values. If this request is made to a device that does not support valuators initialization, a BadMatch error will occur.

If the request succeeds, a status of Success is returned. If another client has the device grabbed, a status of AlreadyGrabbed is returned.

XSetDeviceValuators() can generate a BadLength, BadDevice, BadMatch, or BadValue error.

DIAGNOSTICS

BadDevice
An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*.

BadMatch
This error may occur if an *XSetDeviceValuators()* request is made specifying a device that has no valuators and reports no axes of motion, or if such a request is made specifying a device that does not support valuator initialization.

BadValue
An invalid *first_valuator* or *num_valuators* values was specified.

SEE ALSO

Programming With Xlib

NAME

XSetDeviceMode() – change the mode of a device

SYNOPSIS

XSetDeviceMode (Display **display*, XDevice **device*, int *_mode*)

ARGUMENTS

display Specifies the connection to the X server.
device Specifies the device whose mode is to be changed.
mode Specifies the mode. You can pass Absolute, or Relative.

DESCRIPTION

The *XSetDeviceMode()* request changes the mode of an input device that is capable of reporting either absolute positional information or relative motion information. Not all input devices are capable of reporting motion data, and not all are capable of changing modes from Absolute to Relative.

XSetDeviceMode() can generate a BadDevice or BadMode error.

DIAGNOSTICS**BadDevice**

An invalid device was specified. The specified device does not exist or has not been opened by this client via *XOpenInputDevice()*. This error may also occur if some other client has caused the specified device to become the X keyboard or X pointer device via the *XChangeKeyboardDevice()* or *XChangePointerDevice()* requests.

BadMatch

This error may occur if an *XSetDeviceMode()* request is made specifying a device that has no valuator and reports no axes of motion.

BadMode

An invalid mode was specified. This error will also be returned if the specified device is not capable of supporting the *XSetDeviceMode()* request.

SEE ALSO

Programming With Xlib

XtAddCallback()

NAME

XtAddCallback(), XtAddCallbacks(), XtRemoveCallback(), XtRemoveCallbacks(),
XtRemoveAllCallbacks() – add and remove callback procedures

SYNOPSIS

```
void XtAddCallback (Widget w, String callback_name,  
                  XtCallbackProc callback,  
                  XtPointer client_data)  
void XtAddCallbacks (Widget w, String callback_name,  
                   XtCallbackList callbacks)  
void XtRemoveCallback (Widget w, String callback_name,  
                     XtCallbackProc callback,  
                     XtPointer client_data)  
void XtRemoveCallbacks (Widget w, String callback_name,  
                      XtCallbackList callbacks)  
void XtRemoveAllCallbacks (Widget w, String callback_name)
```

ARGUMENTS

callback Specifies the callback procedure.
callbacks Specifies the null-terminated list of callback procedures and corresponding client data.
callback_name Specifies the callback list to which the procedure is to be appended or deleted.
client_data Specifies the argument that is to be passed to the specified procedure when it is invoked by XtCallbacks or NULL, or the client data to match on the registered callback procedures.
w Specifies the widget.

DESCRIPTION

The *XtAddCallback()* function adds the specified callback procedure to the specified widget's callback list.

The *XtAddCallbacks()* add the specified list of callbacks to the specified widget's callback list.

The *XtRemoveCallback()* function removes a callback only if both the procedure and the client data match.

The *XtRemoveCallbacks()* function removes the specified callback procedures from the specified widget's callback list.

The *XtRemoveAllCallbacks()* function removes all the callback procedures from the specified widget's callback list.

SEE ALSO

XtCallCallbacks()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAddEventHandler()

NAME

XtAddEventHandler(), XtAddRawEventHandler(), XtRemoveEventHandler(),
XtRemoveRawEventHandler() – add and remove event handlers

SYNOPSIS

```
void XtAddEventHandler (Widget w, EventMask event_mask,  
                        Boolean nonmaskable, XtEventHandler proc,  
                        XtPointer client_data)  
void XtAddRawEventHandler (Widget w, EventMask event_mask,  
                           Boolean nonmaskable, XtEventHandler proc,  
                           XtPointer client_data)  
void XtRemoveEventHandler (Widget w, EventMask event_mask,  
                           Boolean nonmaskable, XtEventHandler proc,  
                           XtPointer client_data)  
void XtRemoveRawEventHandler (Widget w, EventMask event_mask,  
                              Boolean nonmaskable, XtEventHandler proc,  
                              XtPointer client_data)
```

ARGUMENTS

client_data
Specifies additional data to be passed to the client's event handler.

event_mask
Specifies the event mask for which to call or unregister this procedure.

nonmaskable
Specifies a Boolean value that indicates whether this procedure should be called or removed on the nonmaskable events (GraphicsExpose, NoExpose, SelectionClear, SelectionRequest, SelectionNotify, ClientMessage, and MappingNotify).

proc
Specifies the procedure that is to be added or removed.

w
Specifies the widget for which this event handler is being registered.

DESCRIPTION

The *XtAddEventHandler()* function registers a procedure with the dispatch mechanism that is to be called when an event that matches the mask occurs on the specified widget. If the procedure is already registered with the same *client_data*, the specified mask is ORed into the existing mask. If the widget is realized, *XtAddEventHandler()* calls *XSelectInput()*, if necessary.

The *XtAddRawEventHandler()* function is similar to *XtAddEventHandler()* except that it does not affect the widget's mask and never causes an *XSelectInput* for its events. Note that the widget might already have those mask bits set because of other nonraw event handlers registered on it.

The *XtAddRawEventHandler()* function is similar to *XtAddEventHandler()* except that it does not affect the widget's mask and never causes an *XSelectInput* for its events. Note that the widget might already have those mask bits set because of other nonraw event handlers registered on it.

The *XtRemoveRawEventHandler()* function stops the specified procedure from receiving the specified events. Because the procedure is a raw event handler, this does not affect the widget's mask and never causes a call on *XSelectInput*.

XtAddEventHandler()

SEE ALSO

XtAppNextEvent(), XtBuildEventMask()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAddExposureToRegion()

NAME

XtAddExposureToRegion() – merge exposure events into a region

SYNOPSIS

```
void XtAddExposureToRegion (XEvent *event, Region region)
```

ARGUMENTS

event Specifies a pointer to the Expose or GraphicsExpose event.
region Specifies the region object (as defined in <X11/Xutil.h>).

DESCRIPTION

The *XtAddExposureToRegion()* function computes the union of the rectangle defined by the exposure event and the specified region. Then, it stores the results back in region. If the event argument is not an Expose or GraphicsExpose event, *XtAddExposureToRegion()* returns without an error and without modifying region.

This function is used by the exposure compression mechanism (see Section 7.9.3).

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAddGrab()

NAME

XtAddGrab(), XtRemoveGrab() – redirect user input to a modal widget

SYNOPSIS

```
void XtAddGrab (Widget w, Boolean exclusive,  
               Boolean spring_loaded)  
void XtRemoveGrab (Widget w)
```

ARGUMENTS

exclusive

Specifies whether user events should be dispatched exclusively to this widget or also to previous widgets in the cascade.

Specifies whether this widget was popped up because the user pressed a pointer button.

w Specifies the widget to add to or remove from the modal cascade.

DESCRIPTION

The *XtAddGrab()* function appends the widget (and associated parameters) to the modal cascade and checks that *exclusive* is True if *spring_loaded* is True. If these are not True, *XtAddGrab()* generates an error.

The modal cascade is used by *XtDispatchEvent()* when it tries to dispatch a user event. When at least one modal widget is in the widget cascade, *XtDispatchEvent()* first determines if the event should be delivered. It starts at the most recent cascade entry and follows the cascade up to and including the most recent cascade entry added with the *exclusive* parameter True.

This subset of the modal cascade along with all descendants of these widgets comprise the active subset. User events that occur outside the widgets in this subset are ignored or remapped. Modal menus with submenus generally add a submenu widget to the cascade with *exclusive* False. Modal dialog boxes that need to restrict user input to the most deeply nested dialog box add a subdialog widget to the cascade with *exclusive* True. User events that occur within the active subset are delivered to the appropriate widget, which is usually a child or further descendant of the modal widget.

Regardless of where on the screen they occur, remap events are always delivered to the most recent widget in the active subset of the cascade that has *spring_loaded* True, if any such widget exists.

The *XtRemoveGrab()* function removes widgets from the modal cascade starting at the most recent widget up to and including the specified widget. It issues an error if the specified widget is not on the modal cascade.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppAddActions()

NAME

XtAppAddActions() – register an action table

SYNOPSIS

```
void XtAppAddActions (XtAppContext app_context, XtActionList actions,  
                      Cardinal num_actions)
```

ARGUMENTS

app_context

Specifies the application context.

actions Specifies the action table to register.

num_args

Specifies the number of entries in this action table.

DESCRIPTION

The *XtAppAddActions()* function adds the specified action table and registers it with the translation manager.

SEE ALSO

XtParseTranslationTable()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppAddConverter()

NAME

XtAppAddConverter() – register resource converter

SYNOPSIS

```
void XtAppAddConverter (XtAppContext app_context, String from_type,  
                        String to_type, XtConverter converter,  
                        XtConvertArgList convert_args, Cardinal num_args)
```

ARGUMENTS

app_context
Specifies the application context.

converter
Specifies the type converter procedure.

convert_args
Specifies how to compute the additional arguments to the converter or NULL.

from_type
Specifies the source type.

num_args
Specifies the number of additional arguments to the converter or zero.

to_type Specifies the destination type.

DESCRIPTION

The *XtAppAddConverter()* registers a the specified resource converter.

SEE ALSO

XtConvert(), *XtStringConversionWarning()*
X Toolkit Intrinsics – C Language Interface
Xlib – C Language X Interface

XtAppAddInput()

NAME

XtAppAddInput(), XtRemoveInput() – register and remove an input source

SYNOPSIS

```
XtInputId XtAppAddInput (XtAppContext app_context, int source,
                        XtPointer condition, XtInputCallbackProc proc,
                        XtPointer client_data)
void XtRemoveInput (XtInputId id)
```

ARGUMENTS

app_context

Specifies the application context that identifies the application.

client_data

Specifies the argument that is to be passed to the specified procedure when input is available.

condition

Specifies the mask that indicates a read, write, or exception condition or some operating system dependent condition.

id

Specifies the ID returned from the corresponding XtAppAddInput call.

proc

Specifies the procedure that is to be called when input is available.

source

Specifies the source file descriptor on a UNIX-based system or other operating system dependent device specification.

DESCRIPTION

The *XtAppAddInput()* function registers with the Intrinsics read routine a new source of events, which is usually file input but can also be file output. Note that file should be loosely interpreted to mean any sink or source of data. *XtAppAddInput()* also specifies the conditions under which the source can generate events. When input is pending on this source, the callback procedure is called.

The legal values for the condition argument are operating-system dependent. On a UNIX-based system, the condition is some union of XtInputReadMask, XtInputWriteMask, and XtInputExceptMask. The *XtRemoveInput()* function causes the Intrinsics read routine to stop watching for input from the input source.

SEE ALSO

XtAppAddTimeOut()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppAddTimeOut()

NAME

XtAppAddTimeOut(), XtRemoveTimeOut() – register and remove timeouts

SYNOPSIS

```
XtIntervalId XtAppAddTimeOut (XtAppContext app_context,
                               unsigned long interval,
                               XtTimerCallbackProc proc,
                               XtPointer client_data)

void XtRemoveTimeOut (XtIntervalId timer)
```

ARGUMENTS

app_context

Specifies the application context for which the timer is to be set.

client_data

Specifies the argument that is to be passed to the specified procedure when called.

interval Specifies the time interval in milliseconds.

proc Specifies the procedure that is to be called when time expires.

timer Specifies the ID for the timeout request to be destroyed.

DESCRIPTION

The *XtAppAddTimeOut()* function creates a timeout and returns an identifier for it. The timeout value is set to *interval*. The callback procedure is called when the time *interval* elapses, and then the timeout is removed.

The *XtRemoveTimeOut()* function removes the timeout. Note that timeouts are automatically removed once they trigger.

SEE ALSO

XtAppAddInput()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppAddWorkProc()

NAME

XtAppAddWorkProc(), XtRemoveWorkProc() – Add and remove background processing procedures

SYNOPSIS

```
XtWorkProcId XtAppAddWorkProc (XtAppContext app_context, XtWorkProc proc,  
                                XtPointer client_data )  
void XtRemoveWorkProc (XtWorkProcId id)
```

ARGUMENTS

app_context

Specifies the application context that identifies the application.

client_data

Specifies the argument that is to be passed to the specified procedure when it is called.

proc

Specifies the procedure that is to be called.

id

Specifies which work procedure to remove.

DESCRIPTION

The *XtAppAddWorkProc()* function adds the specified work procedure for the application identified by *app_context*.

The *XtRemoveWorkProc()* function explicitly removes the specified background work procedure.

SEE ALSO

XtAppNextEvent()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppCreateShell()

NAME

XtAppCreateShell() – create top-level widget instance

SYNOPSIS

```
Widget XtAppCreateShell (String application_name, String application_class,  
                        WidgetClass widget_class, Display *display,  
                        ArgList args, Cardinal num_args)
```

ARGUMENTS

application_class

Specifies the class name of this application.

application_name

Specifies the name of the application instance.

args Specifies the argument list in which to set in the WM_COMMAND property.

display Specifies the display from which to get the resources.

num_args

Specifies the number of arguments in the argument list.

widget_class

Specifies the widget class that the application top-level widget should be.

DESCRIPTION

The *XtAppCreateShell()* function saves the specified application name and application class for qualifying all widget resource specifiers. The application name and application class are used as the left-most components in all widget resource names for this application. *XtAppCreateShell()* should be used to create a new logical application within a program or to create a shell on another display. In the first case, it allows the specification of a new root in the resource hierarchy. In the second case, it uses the resource database associated with the other display.

Note that the widget returned by *XtAppCreateShell()* has the WM_COMMAND property set for session managers (see Chapter 4).

SEE ALSO

XtCreateWidget()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppError()

NAME

XtAppError(), XtAppSetErrorHandler(), XtAppSetWarningHandler(), XtAppWarning()
– low-level error handlers

SYNOPSIS

```
void XtAppError (XtAppContext app_context, String message)
void XtAppSetErrorHandler (XtAppContext app_context,
                           XErrorHandler handler)
void XtAppSetWarningHandler (XtAppContext app_context,
                             XErrorHandler handler)
void XtAppWarning (XtAppContext app_context, String message)
```

ARGUMENTS

app_context Specifies the application context.
message Specifies the nonfatal error message that is to be reported.
handler Specifies the new fatal error procedure, which should not return, or the nonfatal error procedure, which usually returns.
message Specifies the message that is to be reported.

DESCRIPTION

The *XtAppError()* function calls the installed error procedure and passes the specified message.

The *XtAppSetErrorHandler()* function registers the specified procedure, which is called when a fatal error condition occurs.

The *XtAppSetWarningHandler()* registers the specified procedure, which is called when a nonfatal error condition occurs.

The *XtAppWarning()* function calls the installed nonfatal error procedure and passes the specified message.

SEE ALSO

XtAppGetErrorDatabase(), *XtAppErrorMsg()*
X Toolkit Intrinsics – C Language Interface
Xlib – C Language X Interface

XtAppErrorMsg()

NAME

XtAppErrorMsg(), XtAppSetErrorMsgHandler(), XtAppSetWarningMsgHandler(),
XtAppWarningMsg() – high-level error handlers

SYNOPSIS

```
void XtAppErrorMsg (XtAppContext app_context, String name,  
                   String type, String class, String default,  
                   String *params, Cardinal *num_params)  
void XtAppSetErrorMsgHandler (XtAppContext app_context,  
                             XtErrorMsgHandler msg_handler)  
void XtAppSetWarningMsgHandler (XtAppContext app_context,  
                               XtErrorMsgHandler msg_handler)  
void XtAppWarningMsg (XtAppContext app_context, String name,  
                     String type, String class,  
                     String default, String *params,  
                     Cardinal *num_params)
```

ARGUMENTS

app_context Specifies the application context.
class Specifies the resource class.
default Specifies the default message to use.
name Specifies the general kind of error.
type Specifies the detailed name of the error.
msg_handler Specifies the new fatal error procedure, which should not return or the nonfatal error procedure, which usually returns.
num_params Specifies the number of values in the parameter list.
params Specifies a pointer to a list of values to be stored in the message.

DESCRIPTION

The *XtAppErrorMsg()* function calls the high-level error handler and passes the specified information.

The *XtAppSetErrorMsgHandler()* function registers the specified procedure, which is called when a fatal error occurs.

The *XtAppSetWarningMsgHandler()* function registers the specified procedure, which is called when a nonfatal error condition occurs.

The *XtAppWarningMsg()* function calls the high-level error handler and passes the specified information.

SEE ALSO

XtAppGetErrorDatabase(), *XtAppError()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppGetErrorDatabase()

NAME

XtAppGetErrorDatabase(), XtAppGetErrorDatabaseText() – obtain error database

SYNOPSIS

```
XrmDatabase *XtAppGetErrorDatabase (XtAppContext app_context)
void XtAppGetErrorDatabaseText (XtAppContext app_context,
                                char *name,
                                char *type, char *class,
                                char *default, char *buffer_return,
                                int nbytes, XrmDatabase database)
```

ARGUMENTS

app_context Specifies the application context.

buffer_return Specifies the buffer into which the error message is to be returned.

class Specifies the resource class of the error message.

database Specifies the name of the alternative database that is to be used or NULL if the application's database is to be used.

default Specifies the default message to use.

name

type Specifies the name and type that are concatenated to form the resource name of the error message.

nbytes Specifies the size of the buffer in bytes.

DESCRIPTION

The *XtAppGetErrorDatabase()* function returns the address of the error database. The Intrinsics do a lazy binding of the error database and do not merge in the database file until the first call to *XtAppGetErrorDatabaseText()*.

The *XtAppGetErrorDatabaseText()* returns the appropriate message from the error database or returns the specified default message if one is not found in the error database.

SEE ALSO

XtAppError(), *XtAppErrorMsg()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppGetSelectionTimeout()

NAME

XtAppGetSelectionTimeout(), XtAppSetSelectionTimeout() – set and obtain selection timeout values

SYNOPSIS

```
unsigned long XtAppGetSelectionTimeout (XtAppContext app_context)  
void XtAppSetSelectionTimeout (XtAppContext app_context,  
                               unsigned long timeout)
```

ARGUMENTS

app_context

Specifies the application context.

timeout Specifies the selection timeout in milliseconds.

DESCRIPTION

The *XtAppGetSelectionTimeout()* function returns the current selection timeout value, in milliseconds. The selection timeout is the time within which the two communicating applications must respond to one another. The initial timeout value is set by the selectionTimeout application resource, or, if selectionTimeout is not specified, it defaults to five seconds.

The *XtAppSetSelectionTimeout()* function sets the Intrinsics's selection timeout mechanism. Note that most applications should not set the selection timeout.

SEE ALSO

XtOwnSelection()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtAppNextEvent()

NAME

XtAppNextEvent(), XtAppPending(), XtAppPeekEvent(), XtAppProcessEvent(),
XtDispatchEvent(), XtAppMainLoop() – query and process events and input

SYNOPSIS

```
void XtAppNextEvent (XtAppContext app_context, XEvent *event_return)
Boolean XtAppPeekEvent (XtAppContext app_context, XEvent *event_return)
XtInputMask XtAppPending (XtAppContext app_context)
void XtAppProcessEvent (XtAppContext app_context, XtInputMask mask)
Boolean XtDispatchEvent (XEvent *event)
void XtAppMainLoop (XtAppContext app_context)
```

ARGUMENTS

app_context Specifies the application context that identifies the application.

event Specifies a pointer to the event structure that is to be dispatched to the appropriate event handler.

event_return Returns the event information to the specified event structure.

mask Specifies what types of events to process. The mask is the bitwise inclusive OR of any combination of XtIMXEvent, XtIMTimer, and XtIMAlternateInput. As a convenience, the X Toolkit defines the symbolic name XtIMAll to be the bitwise inclusive OR of all event types.

DESCRIPTION

If no input is on the X input queue, *XtAppNextEvent()* flushes the X output buffer and waits for an event while looking at the other input sources and timeout values and calling any callback procedures triggered by them. This wait time can be used for background processing (see Section 7.8).

If there is an event in the queue, *XtAppPeekEvent()* fills in the event and returns a nonzero value. If no X input is on the queue, *XtAppPeekEvent()* flushes the output buffer and blocks until input is available (possibly calling some timeout callbacks in the process). If the input is an event, *XtAppPeekEvent()* fills in the event and returns a nonzero value. Otherwise, the input is for an alternate input source, and *XtAppPeekEvent()* returns zero.

The *XtAppPending()* function returns a nonzero value if there are events pending from the X server, timer pending, or other input sources pending. The value returned is a bit mask that is the OR of XtIMXEvent, XtIMTimer, and XtIMAlternateInput (see *XtAppProcessEvent()*). If there are no events pending, *XtAppPending()* flushes the output buffer and returns zero.

The *XtAppProcessEvent()* function processes one timer, alternate input, or X event. If there is nothing of the appropriate type to process, *XtAppProcessEvent()* blocks until there is. If there is more than one type of thing available to process, it is undefined which will get processed. Usually, this procedure is not called by client applications (see *XtAppMainLoop()*). *XtAppProcessEvent()* processes timer events by calling any appropriate timer callbacks, alternate input by calling any appropriate alternate input callbacks, and X events by calling *XtDispatchEvent()*.

When an X event is received, it is passed to *XtDispatchEvent()*, which calls the appropriate event handlers and passes them the widget, the event, and client-specific

XtAppNextEvent()

data registered with each procedure. If there are no handlers for that event registered, the event is ignored and the dispatcher simply returns. The order in which the handlers are called is undefined.

The *XtDispatchEvent()* function sends those events to the event handler functions that have been previously registered with the dispatch routine. *XtDispatchEvent()* returns True if it dispatched the event to some handler and False if it found no handler to dispatch the event to. The most common use of *XtDispatchEvent()* is to dispatch events acquired with the *XtAppNextEvent()* procedure. However, it also can be used to dispatch user-constructed events. *XtDispatchEvent()* also is responsible for implementing the grab semantics for *XtAddGrab()*.

The *XtAppMainLoop()* function first reads the next incoming X event by calling *XtAppNextEvent()* and then it dispatches the event to the appropriate registered procedure by calling *XtDispatchEvent()*. This constitutes the main loop of X Toolkit applications, and, as such, it does not return. Applications are expected to exit in response to some user action. There is nothing special about *XtAppMainLoop()*; it is simply an infinite loop that calls *XtAppNextEvent()* and then *XtDispatchEvent()*.

Applications can provide their own version of this loop, which tests some global termination flag or tests that the number of top-level widgets is larger than zero before circling back to the call to *XtAppNextEvent()*.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtBuildEventMask()

NAME

XtBuildEventMask() – retrieve a widget’s event mask

SYNOPSIS

EventMask XtBuildEventMask (Widget w)

ARGUMENTS

w Specifies the widget.

DESCRIPTION

The *XtBuildEventMask()* function returns the event mask representing the logical OR of all event masks for event handlers registered on the widget with *XtAddEventHandler()* and all event translations, including accelerators, installed on the widget. This is the same event mask stored into the **XSetWindowAttributes** structure by *XtRealizeWidget()* and sent to the server when event handlers and translations are installed or removed on the realized widget.

SEE ALSO

XtAddEventHandler()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtCallAcceptFocus()

NAME

XtCallAcceptFocus() – call a widget’s `accept_focus` procedure

SYNOPSIS

Boolean XtCallAcceptFocus (Widget *w*, Time **time*)

ARGUMENTS

time Specifies the X time of the event that is causing the accept focus.
w Specifies the widget.

DESCRIPTION

The *XtCallAcceptFocus()* function calls the specified widget’s *accept_focus* procedure, passing it the specified widget and time, and returns what the *accept_focus* procedure returns. If *accept_focus* is NULL, *XtCallAcceptFocus()* returns False .

SEE ALSO

XtSetKeyboardFocus()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtCallCallbacks()

NAME

XtCallCallbacks(), XtHasCallbacks() – process callbacks

SYNOPSIS

```
void XtCallCallbacks (Widget w, String callback_name,  
                    XtPointer call_data)  
typedef enum {XtCallbackNoList,  
             XtCallbackHasNone,  
             XtCallbackHasSome} XtCallbackStatus  
  
XtCallbackStatus XtHasCallbacks (Widget w, String callback_name)
```

ARGUMENTS

callback_name
Specifies the callback list to be executed or checked.

call_data
Specifies a callback-list specific data value to pass to each of the callback procedure in the list.

w
Specifies the widget.

DESCRIPTION

The *XtCallCallbacks()* function calls each procedure that is registered in the specified widget's callback list.

The *XtHasCallbacks()* function first checks to see if the widget has a callback list identified by *callback_name*. If the callback list does not exist, *XtHasCallbacks()* returns XtCallbackNoList. If the callback list exists but is empty, it returns XtCallbackHasNone. If the callback list exists and has at least one callback registered, it returns XtCallbackHasSome.

SEE ALSO

XtAddCallback()
X Toolkit Intrinsics – C Language Interface
Xlib – C Language X Interface

NAME

XtClass(), XtSuperClass(), XtIsSubclass(), XtCheckSubclass(), XtIsComposite(), XtIsManaged() – obtain and verify a widget's class

SYNOPSIS

```
WidgetClass XtClass (Widget w)
WidgetClass XtSuperclass (Widget w)
Boolean XtIsSubclass (Widget w, WidgetClass widget_class)
void XtCheckSubclass (Widget w, WidgetClass widget_class,
                     String message)
Boolean XtIsComposite (Widget w)
Boolean XtIsManaged (Widget w)
```

ARGUMENTS

w Specifies the widget.
widget_class Specifies the widget class.
message Specifies the message that is to be used.

DESCRIPTION

The *XtClass()* function returns a pointer to the widget's class structure.

The *XtSuperclass()* function returns a pointer to the widget's superclass class structure.

The *XtIsSubclass()* function returns True if the class of the specified widget is equal to or is a subclass of the specified widget class. The specified widget can be any number of subclasses down the chain and need not be an immediate subclass of the specified widget class. Composite widgets that need to restrict the class of the items they contain can use *XtIsSubclass()* to find out if a widget belongs to the desired class of objects.

The *XtCheckSubclass()* macro determines if the class of the specified widget is equal to or is a subclass of the specified widget class. The widget can be any number of subclasses down the chain and need not be an immediate subclass of the specified widget class. If the specified widget is not a subclass, *XtCheckSubclass()* constructs an error message from the supplied message, the widget's actual class, and the expected class and calls *XtErrorMsg()*. *XtCheckSubclass()* should be used at the entry point of exported routines to ensure that the client has passed in a valid widget class for the exported operation.

XtCheckSubclass() is only executed when the widget has been compiled with the compiler symbol *DEBUG* defined; otherwise, it is defined as the empty string and generates no code.

The *XtIsComposite()* function is a convenience function that is equivalent to *XtIsSubclass()* with *compositeWidgetClass* specified.

The *XtIsManaged()* macro (for widget programmers) or function (for application programmers) returns True if the specified child widget is managed or False if it is not.

SEE ALSO

XtAppErrorMsg(), *XtDisplay()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtConfigureWidget()

NAME

XtConfigureWidget(), XtMoveWidget(), XtResizeWidget() – move and resize widgets

SYNOPSIS

```
void XtConfigureWidget (Widget w, Position x, Position y,  
                        Dimension width, Dimension height,  
                        Dimension border_width)  
void XtMoveWidget (Widget w, Position x, Position y)  
void XtResizeWidget (Widget w, Dimension width, Dimension height,  
                    Dimension border_width)  
void XtResizeWindow (Widget w)
```

ARGUMENTS

width
height
border_width Specify the new widget size.
w Specifies the widget.
x
y Specify the new widget x and y coordinates.

DESCRIPTION

The *XtConfigureWidget()* function returns immediately if the specified geometry fields are the same as the old values. Otherwise, *XtConfigureWidget()* writes the new *x*, *y*, *width*, *height*, and *border_width* values into the widget and, if the widget is realized, makes an Xlib *XConfigureWindow()* call on the widget's window.

If either the new *width* or *height* is different from its old value, *XtConfigureWidget()* calls the widget's resize procedure to notify it of the size change; otherwise, it simply returns.

The *XtMoveWidget()* function returns immediately if the specified geometry fields are the same as the old values. Otherwise, *XtMoveWidget()* writes the new *x* and *y* values into the widget and, if the widget is realized, issues an Xlib *XMoveWindow()* call on the widget's window.

The *XtResizeWidget()* function returns immediately if the specified geometry fields are the same as the old values. Otherwise, *XtResizeWidget()* writes the new *width*, *height*, and *border_width* values into the widget and, if the widget is realized, issues an *XConfigureWindow()* call on the widget's window.

If the new *width* or *height* are different from the old values, *XtResizeWidget()* calls the widget's resize procedure to notify it of the size change.

The *XtResizeWindow()* function calls the *XConfigureWindow()* Xlib function to make the window of the specified widget match its width, height, and border width. This request is done unconditionally because there is no way to tell if these values match the current values. Note that the widget's resize procedure is not called.

There are very few times to use *XtResizeWindow()*; instead, you should use *XtResizeWidget()*.

SEE ALSO

XtMakeGeometryRequest(), *XtQueryGeometry()*

XtConfigureWidget()

X Toolkit Intrinsic – C Language Interface

Xlib – C Language X Interface

XtConvert()

NAME

XtConvert(), XtDirectConvert() – invoke resource converters

SYNOPSIS

```
void XtConvert (Widget w, String from_type, XrmValuePtr from,
               String to_type, XrmValuePtr to_return)
void XtDirectConvert (XtConverter converter, XrmValuePtr args,
                    Cardinal num_args, XrmValuePtr from,
                    XrmValuePtr to_return)
```

ARGUMENTS

args Specifies the argument list that contains the additional arguments needed to perform the conversion (often NULL).

converter Specifies the conversion procedure that is to be called.

from Specifies the value to be converted.

from_type Specifies the source type.

num_args Specifies the number of additional arguments (often zero).

to_type Specifies the destination type.

to_return Returns the converted value.

w Specifies the widget to use for additional arguments (if any are needed).

DESCRIPTION

The *XtConvert()* function looks up the type converter registered to convert *from_type* to *to_type*, computes any additional arguments needed, and then calls *XtDirectConvert()*.

The *XtDirectConvert()* function looks in the converter cache to see if this conversion procedure has been called with the specified arguments. If so, it returns a descriptor for information stored in the cache; otherwise, it calls the converter and enters the result in the cache.

Before calling the specified converter, *XtDirectConvert()* sets the return value size to zero and the return value address to NULL. To determine if the conversion was successful, the client should check *to_return.address* for non-NULL.

SEE ALSO

XtAppAddConverter(), *XtStringConversionWarning()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtCreateApplicationContext()

NAME

XtCreateApplicationContext(), XtDestroyApplicationContext(),
XtWidgetToApplicationContext(), XtToolkitInitialize() – create, destroy, and obtain an
application context

SYNOPSIS

```
XtAppContext XtCreateApplicationContext (void)
void XtDestroyApplicationContext (XtAppContext app_context)
XtAppContext XtWidgetToApplicationContext (Widget w)
void XtToolkitInitialize (void)
```

ARGUMENTS

app_context Specifies the application context.
w Specifies the widget .

DESCRIPTION

The *XtCreateApplicationContext()* function returns an application context, which is an opaque type. Every application must have at least one application context.

The *XtDestroyApplicationContext()* function destroys the specified application context as soon as it is safe to do so. If called from within an event dispatch (for example, a callback procedure), *XtDestroyApplicationContext()* does not destroy the application context until the dispatch is complete.

The *XtWidgetToApplicationContext()* function returns the application context for the specified widget.

The semantics of calling *XtToolkitInitialize()* more than once are undefined.

SEE ALSO

XtDisplayInitialize()
X Toolkit Intrinsics – C Language Interface
Xlib – C Language X Interface

XtCreatePopupShell()

NAME

XtCreatePopupShell() – create a popup shell

SYNOPSIS

```
Widget XtCreatePopupShell (String name, WidgetClass widget_class,  
                           Widget parent, ArgList args,  
                           Cardinal num_args)
```

ARGUMENTS

args Specifies the argument list to override the resource defaults.
name Specifies the text name for the created shell widget.
num_args Specifies the number of arguments in the argument list.
parent Specifies the parent widget.
widget_class Specifies the widget class pointer for the created shell widget.

DESCRIPTION

The *XtCreatePopupShell()* function ensures that the specified class is a subclass of Shell and, rather than using *insert_child* to attach the widget to the parent's children list, attaches the shell to the parent's pop-ups list directly.

A spring-loaded pop-up invoked from a translation table already must exist at the time that the translation is invoked, so the translation manager can find the shell by name. Pop-ups invoked in other ways can be created “on-the-fly” when the pop-up actually is needed. This delayed creation of the shell is particularly useful when you pop up an unspecified number of pop-ups. You can look to see if an appropriate unused shell (that is, not currently popped up) exists and create a new shell if needed.

SEE ALSO

XtCreateWidget(), *XtPopdown()*, *XtPopup()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtCreateWidget()

NAME

XtCreateWidget(), XtCreateManagedWidget(), XtDestroyWidget() – create and destroy widgets

SYNOPSIS

```
Widget XtCreateWidget (String name, WidgetClass widget_class,
                      Widget parent, ArgList args,
                      Cardinal num_args)
Widget XtCreateManagedWidget (String name, WidgetClass widget_class,
                              Widget parent, ArgList args,
                              Cardinal num_args)
void XtDestroyWidget (Widget w)
```

ARGUMENTS

args Specifies the argument list to override the resource defaults.
name Specifies the resource name for the created widget, which is used for retrieving resources and, for that reason, should not be the same as any other widget that is a child of same parent.
num_args Specifies the number of arguments in the argument list.
parent Specifies the parent widget.
w Specifies the widget.
widget_class Specifies the widget class pointer for the created widget.

DESCRIPTION

The *XtCreateWidget()* function performs much of the boilerplate operations of widget creation:

- Checks to see if the *class_initialize* procedure has been called for this class and for all superclasses and, if not, calls those necessary in a superclass-to-subclass order.
- Allocates memory for the widget instance.
- If the parent is a subclass of *constraintWidgetClass*, it allocates memory for the parent's constraints and stores the address of this memory into the constraints field.
- Initializes the core nonresource data fields (for example, parent and visible).
- Initializes the resource fields (for example, background_pixel) by using the resource lists specified for this class and all superclasses.
- If the parent is a subclass of *constraintWidgetClass()*, it initializes the resource fields of the constraints record by using the constraint resource list specified for the parent's class and all superclasses up to *constraintWidgetClass()*.
- Calls the initialize procedures for the widget by starting at the *Core* initialize procedure on down to the widget's initialize procedure.
- If the parent is a subclass of *compositeWidgetClass()*, it puts the widget into its parent's children list by calling its parent's *insert_child* procedure. For further information, see Section 3.5.
- If the parent is a subclass of *constraintWidgetClass()*, it calls the constraint initialize procedures, starting at *constraintWidgetClass* on down to the parent's constraint

initialize procedure.

Note that you can determine the number of arguments in an argument list by using the *XtNumber()* macro. For further information, see Section 11.1.

The *XtCreateManagedWidget()* function is a convenience routine that calls *XtCreateWidget()* and *XtManageChild()*.

The *XtDestroyWidget()* function provides the only method of destroying a widget, including widgets that need to destroy themselves. It can be called at any time, including from an application callback routine of the widget being destroyed. This requires a two-phase destroy process in order to avoid dangling references to destroyed widgets.

In phase one, *XtDestroyWidget()* performs the following:

- If the *being_destroyed* field of the widget is True, it returns immediately.
- Recursively descends the widget tree and sets the *being_destroyed* field to True for the widget and all children.
- Adds the widget to a list of widgets (the destroy list) that should be destroyed when it is safe to do so.

Entries on the destroy list satisfy the invariant that if w2 occurs after w1 on the destroy list then w2 is not a descendent of w1. (A descendant refers to both normal and pop-up children.)

Phase two occurs when all procedures that should execute as a result of the current event have been called (including all procedures registered with the event and translation managers), that is, when the current invocation of *XtDispatchEvent()* is about to return or immediately if not in *XtDispatchEvent()*.

In phase two, *XtDestroyWidget()* performs the following on each entry in the destroy list:

- Calls the destroy callback procedures registered on the widget (and all descendants) in post-order (it calls children callbacks before parent callbacks).
- If the widget's parent is a subclass of *compositeWidgetClass* and if the parent is not being destroyed, it calls *XtUnmanageChild()* on the widget and then calls the widget's parent's *delete_child* procedure (see Section 3.4).
- If the widget's parent is a subclass of *constraintWidgetClass*, it calls the constraint destroy procedure for the parent, then the parent's superclass, until finally it calls the constraint destroy procedure for *constraintWidgetClass*.
- Calls the destroy methods for the widget (and all descendants) in post-order. For each such widget, it calls the destroy procedure declared in the widget class, then the destroy procedure declared in its superclass, until finally it calls the destroy procedure declared in the Core class record.
- Calls *XDestroyWindow()* if the widget is realized (that is, has an X window). The server recursively destroys all descendant windows.
- Recursively descends the tree and deallocates all pop-up widgets, constraint records, callback lists and, if the widget is a subclass of *compositeWidgetClass*, children.

SEE ALSO

XtAppCreateShell(), *XtCreatePopupShell()*

X Toolkit Intrinsic – C Language Interface

Xlib – C Language X Interface

XtCreateWindow()

NAME

XtCreateWindow() – window creation convenience function

SYNOPSIS

```
void XtCreateWindow (Widget w, unsigned int window_class,  
                    Visual *visual, XtValueMask value_mask,  
                    XSetWindowAttributes *attributes)
```

ARGUMENTS

attributes

Specifies the window attributes to use in the *XCreateWindow()* call.

value_mask

Specifies which attribute fields to use.

visual

Specifies the visual type (usually CopyFromParent).

w

Specifies the widget that is used to set the x,y coordinates and so on.

window_class

Specifies the Xlib window class (for example, InputOutput, InputOnly, or CopyFromParent).

DESCRIPTION

The *XtCreateWindow()* function calls the Xlib *XCreateWindow()* function with values from the widget structure and the passed parameters. Then, it assigns the created window to the widget's window field.

XtCreateWindow() evaluates the following fields of the *Core()* widget structure:

- *depth*
- *screen*
- *parent->core.window*
- *x*
- *y*
- *width*
- *height*
- *border_width*

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtDisplay()

NAME

XtDisplay(), XtParent(), XtScreen(), XtWindow() – obtain window information about a widget

SYNOPSIS

```
Display *XtDisplay (Widget w)
Widget XtParent (Widget w)
Screen *XtScreen (Widget w)
Window XtWindow (Widget w)
```

ARGUMENTS

w Specifies the widget.

DESCRIPTION

XtDisplay() returns the display pointer for the specified widget.

XtParent() returns the parent widget for the specified widget.

XtScreen() returns the screen pointer for the specified widget.

XtWindow() returns the window of the specified widget.

SEE ALSO

XtClass()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtDisplayInitialize()

NAME

XtDisplayInitialize(), XtOpenDisplay(), XtDatabase(), XtCloseDisplay() – initialize, open, or close a display

SYNOPSIS

```
void XtToolkitInitialize (void)
void XtDisplayInitialize (XtAppContext app_context, Display *display,
                          String application_name,
                          String application_class,
                          XrmOptionDescRec *options,
                          Cardinal num_options,
                          Cardinal *argc, String *argv)
Display *XtOpenDisplay (XtAppContext app_context, String display_string,
                        String application_name,
                        String application_class,
                        XrmOptionDescRec *options,
                        Cardinal num_options,
                        Cardinal *argc, String *argv)
void XtCloseDisplay (Display *display)
XrmDatabase XtDatabase (Display *display)
```

ARGUMENTS

argc Specifies a pointer to the number of command line parameters.

argv Specifies the command line parameters.

app_context
Specifies the application context.

application_class
Specifies the class name of this application, which usually is the generic name for all instances of this application.

application_name
Specifies the name of the application instance.

display Specifies the display. Note that a display can be in at most one application context.

num_options
Specifies the number of entries in the options list.

options Specifies how to parse the command line for any application-specific resources. The *options* argument is passed as a parameter to *XrmParseCommand()*. For further information, see *Xlib – C Language X Interface*.

DESCRIPTION

The *XtDisplayInitialize()* function builds the resource database, calls the Xlib *XrmParseCommand()* function to parse the command line, and performs other per display initialization. After *XrmParseCommand()* has been called, *argc* and *argv* contain only those parameters that were not in the standard option table or in the table specified by the options argument. If the modified **Bad.xcac macro** is not zero, most applications simply print out the modified *argv* along with a message listing the allowable options. On UNIX-based systems, the application name is usually the final component of *argv[0]*. If the synchronize resource is True for the specified application, *XtDisplayInitialize()* calls the Xlib *XSynchronize()* function to put Xlib into synchronous mode for this display connection. If the reverseVideo resource is True, the Intrinsics exchange *XtDefaultForeground()* and *XtDefaultBackground()* for widgets created on this display.

XtDisplayInitialize()

(See Section 9.6.1).

The *XtOpenDisplay()* function calls *XOpenDisplay()* the specified display name. If *display_string* is NULL, *XtOpenDisplay()* uses the current value of the **—display** option specified in *argv* and if no display is specified in *argv*, uses the user's default display (on UNIX-based systems, this is the value of the *DISPLAY* environment variable).

If this succeeds, it then calls *XtDisplayInitialize()* and pass it the opened display and the value of the **—name** option specified in *argv* as the application name. If no name option is specified, it uses the application name passed to *XtOpenDisplay()*. If the application name is NULL, it uses the last component of *argv*[0]. *XtOpenDisplay()* returns the newly opened display or NULL if it failed.

XtOpenDisplay() is provided as a convenience to the application programmer.

The *XtCloseDisplay()* function closes the specified display as soon as it is safe to do so. If called from within an event dispatch (for example, a callback procedure), *XtCloseDisplay()* does not close the display until the dispatch is complete. Note that applications need only call *XtCloseDisplay()* if they are to continue executing after closing the display; otherwise, they should call *XtDestroyApplicationContext()* or just exit.

The *XtDatabase()* function returns the fully merged resource database that was built by *XtDisplayInitialize()* associated with the display that was passed in. If this display has not been initialized by *XtDisplayInitialize()*, the results are not defined.

SEE ALSO

XtAppCreateShell(), *XtCreateApplicationContext()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtGetGC()

NAME

XtGetGC(), XtReleaseGC() – obtain and destroy a sharable GC

SYNOPSIS

```
GC XtGetGC (Widget w, XtGCMask value_mask, XGCValues *values)
void XtReleaseGC (Widget w, GC gc)
```

ARGUMENTS

gc Specifies the GC to be deallocated.
values Specifies the actual values for this GC.
value_mask Specifies which fields of the values are specified.
w Specifies the widget.

DESCRIPTION

The *XtGetGC()* function returns a sharable, read-only GC. The parameters to this function are the same as those for *XCreateGC()* except that a widget is passed instead of a display. *XtGetGC()* shares only GCs in which all values in the GC returned by *XCreateGC()* are the same. In particular, it does not use the *value_mask* provided to determine which fields of the GC a widget considers relevant. The *value_mask* is used only to tell the server which fields should be filled in with widget data and which it should fill in with default values. For further information about *value_mask* and *values*, see *XCreateGC()* in the *Xlib – C Language X Interface*.

The *XtReleaseGC()* function deallocate the specified shared GC.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtGetResourceList()

NAME

XtGetResourceList() – obtain resource list

SYNOPSIS

```
void XtGetResourceList (WidgetClass class,  
                        XtResourceList *resources_return,  
                        Cardinal *num_resources_return)
```

ARGUMENTS

num_resources_return

Specifies a pointer to where to store the number of entries in the resource list.

resources_return

Specifies a pointer to where to store the returned resource list. The caller must free this storage using *XtFree()* when done with it.

widget_class

Specifies the widget class.

DESCRIPTION

If it is called before the widget class is initialized (that is, before the first widget of that class has been created), *XtGetResourceList()* returns the resource list as specified in the widget class record. If it is called after the widget class has been initialized, *XtGetResourceList()* returns a merged resource list that contains the resources for all superclasses.

SEE ALSO

XtGetSubresources(), *XtOffset()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtGetSelectionValue()

NAME

XtGetSelectionValue(), XtGetSelectionValues() – obtain selection values

SYNOPSIS

```
void XtGetSelectionValue (Widget w, Atom selection,
                          Atom target,
                          XtSelectionCallbackProc callback,
                          XtPointer client_data, Time time)
void XtGetSelectionValues (Widget w, Atom selection,
                          Atom *targets, int count,
                          XtSelectionCallbackProc callback,
                          XtPointer client_data, Time time)
```

ARGUMENTS

callback Specifies the callback procedure that is to be called when the selection value has been obtained.

client_data Specifies the argument that is to be passed to the specified procedure when it is called.

client_data Specifies the client data (one for each target type) that is passed to the callback procedure when it is called for that target.

count Specifies the length of the *targets* and *client_data* lists.

selection Specifies the particular selection desired (that is, primary or secondary).

target Specifies the type of the information that is needed about the selection.

targets Specifies the types of information that is needed about the selection.

time Specifies the timestamp that indicates when the selection value is desired.

w Specifies the widget that is making the request.

DESCRIPTION

The *XtGetSelectionValue()* function requests the value of the selection that has been converted to the *target* type. The specified callback will be called some time after *XtGetSelectionValue()* is called; in fact, it may be called before or after *XtGetSelectionValue()* returns.

The *XtGetSelectionValues()* function is similar to *XtGetSelectionValue()* except that it takes a list of target types and a list of client data and obtains the current value of the selection converted to each of the targets. The effect is as if each target were specified in a separate call to *XtGetSelectionValue()*. The callback is called once with the corresponding client data for each target. *XtGetSelectionValues()* does guarantee that all the conversions will use the same selection value because the ownership of the selection cannot change in the middle of the list, as would be when calling *XtGetSelectionValue()* repeatedly.

SEE ALSO

XtAppGetSelectionTimeout(), *XtOwnSelection()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtGetSubresources()

NAME

XtGetSubresources(), XtGetApplicationResources() – obtain subresources or application resources

SYNOPSIS

```
void XtGetSubresources (Widget w, XtPointer base, String name,
                        String class, XtResourceList resources,
                        Cardinal num_resources, ArgList args,
                        Cardinal num_args)
void XtGetApplicationResources (Widget w, XtPointer base,
                                XtResourceList resources,
                                Cardinal num_resources,
                                ArgList args, Cardinal num_args)
```

ARGUMENTS

<i>args</i>	Specifies the argument list to override resources obtained from the resource database.
<i>base</i>	Specifies the base address of the subpart data structure where the resources should be written.
<i>class</i>	Specifies the class of the subpart.
<i>name</i>	Specifies the name of the subpart.
<i>num_args</i>	Specifies the number of arguments in the argument list.
<i>num_resources</i>	Specifies the number of resources in the resource list.
<i>resources</i>	Specifies the resource list for the subpart.
<i>w</i>	Specifies the widget that wants resources for a subpart or that identifies the resource database to search.

DESCRIPTION

The *XtGetSubresources()* function constructs a name/class list from the application name/class, the name/classes of all its ancestors, and the widget itself. Then, it appends to this list the *name/class* pair passed. The resources are fetched from the argument list, the resource database, or the default values in the resource list. Then, they are copied into the subpart record. If *args* is NULL, *num_args* must be zero. However, if *num_args* is zero, the argument list is not referenced.

The *XtGetApplicationResources()* function first uses the passed widget, which is usually an application shell, to construct a resource name and class list. Then, it retrieves the resources from the argument list, the resource database, or the resource list default values. After adding base to each address, *XtGetApplicationResources()* copies the resources into the address given in the resource list. If *args* is NULL, *num_args* must be zero. However, if *num_args* is zero, the argument list is not referenced. The portable way to specify application resources is to declare them as members of a structure and pass the address of the structure as the base argument.

SEE ALSO

XtGetResourceList()

X Toolkit Intrinsics – C Language Interface

XtGetSubresources()

Xlib – C Language X Interface

XtMakeGeometryRequest()

NAME

XtMakeGeometryRequest(), XtMakeResizeRequest() – make geometry manager request

SYNOPSIS

```
XtGeometryResult XtMakeGeometryRequest (Widget w,  
                                         XtWidgetGeometry *request,  
                                         XtWidgetGeometry *reply_return)  
XtGeometryResult XtMakeResizeRequest (Widget w, Dimension width,  
                                       Dimension height,  
                                       Dimension *width_return,  
                                       Dimension *height_return)
```

ARGUMENTS

reply_return Returns the allowed widget size or may be NULL if the requesting widget is not interested in handling XtGeometryAlmost .

request Specifies the desired widget geometry (size, position, border width, and stacking order).

w Specifies the widget that is making the request.

width_return
height_return Return the allowed widget width and height.

DESCRIPTION

Depending on the condition, *XtMakeGeometryRequest()* performs the following:

- If the widget is unmanaged or the widget's parent is not realized, it makes the changes and returns XtGeometryYes.
- If the parent is not a subclass of compositeWidgetClass or the parent's *geometry_manager* is NULL, it issues an error.
- If the widget's *being_destroyed* field is True, it returns XtGeometryNo.
- If the widget *x*, *y*, *width*, *height* and *border_width* fields are all equal to the requested values, it returns XtGeometryYes; otherwise, it calls the parent's *geometry_manager* procedure with the given parameters.
- If the parent's geometry manager returns XtGeometryYes and if *XtCWQueryOnly* is not set in the request_mode and if the widget is realized, *XtMakeGeometryRequest()* calls the *XConfigureWindow()* Xlib function to reconfigure the widget's window (set its size, location, and stacking order as appropriate).
- If the geometry manager returns XtGeometryDone, the change has been approved and actually has been done. In this case, *XtMakeGeometryRequest()* does no configuring and returns XtGeometryYes. *XtMakeGeometryRequest()* never returns XtGeometryDone.

Otherwise, *XtMakeGeometryRequest()* returns the resulting value from the parent's geometry manager.

Children of primitive widgets are always unmanaged; thus, *XtMakeGeometryRequest()* always returns XtGeometryYes when called by a child of a primitive widget.

XtMakeGeometryRequest()

The *XtMakeResizeRequest()* function, a simple interface to *XtMakeGeometryRequest()*, creates a **XtWidgetGeometry** structure and specifies that width and height should change. The geometry manager is free to modify any of the other window attributes (position or stacking order) to satisfy the resize request. If the return value is *XtGeometryAlmost*, *width_return* and *height_return* contain a compromise width and height. If these are acceptable, the widget should immediately make an **XtMakeResizeRequest** and request that the compromise width and height be applied. If the widget is not interested in *XtGeometryAlmost* replies, it can pass NULL for *width_return* and *height_return*.

SEE ALSO

XtConfigureWidget(), *XtQueryGeometry()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

NAME

XtMalloc(), XtCalloc(), XtRealloc(), XtFree(), XtNew(), XtNewString() – memory management functions

SYNOPSIS

```
char *XtMalloc (Cardinal size)
char *XtCalloc (Cardinal num, Cardinal size)
char *XtRealloc (char *ptr, Cardinal num)
void XtFree (char *ptr)
type *XtNew (type)
String XtNewString (String string)
```

ARGUMENTS

<i>num</i>	Specifies the number of bytes or array elements.
<i>ptr</i>	Specifies a pointer to the old storage or to the block of storage that is to be freed.
<i>size</i>	Specifies the size of an array element (in bytes) or the number of bytes desired.
<i>string</i>	Specifies a previously declared string.
<i>type</i>	Specifies a previously declared data type.

DESCRIPTION

The *XtMalloc()* functions returns a pointer to a block of storage of at least the specified *size* bytes. If there is insufficient memory to allocate the new block, *XtMalloc()* calls *XtErrorMsg()*.

The *XtCalloc()* function allocates space for the specified number of array elements of the specified size and initializes the space to zero. If there is insufficient memory to allocate the new block, *XtCalloc()* calls *XtErrorMsg()*.

The *XtRealloc()* function changes the size of a block of storage (possibly moving it). Then, it copies the old contents (or as much as will fit) into the new block and frees the old block. If there is insufficient memory to allocate the new block, *XtRealloc()* calls *XtErrorMsg()*. If *ptr* is NULL, *XtRealloc()* allocates the new storage without copying the old contents; that is, it simply calls *XtMalloc()*.

The *XtFree()* function returns storage and allows it to be reused. If *ptr* is NULL, *XtFree()* returns immediately.

XtNew() returns a pointer to the allocated storage. If there is insufficient memory to allocate the new block, *XtNew()* calls *XtErrorMsg()*. *XtNew()* is a convenience macro that calls *XtMalloc()* with the following arguments specified:

```
((type *) XtMalloc((unsigned) sizeof(type)))
```

XtNewString() returns a pointer to the allocated storage. If there is insufficient memory to allocate the new block, *XtNewString()* calls *XtErrorMsg()*. *XtNewString()* is a convenience macro that calls *XtMalloc()* with the following arguments specified:

```
(strcpy(XtMalloc((unsigned) strlen(str) + 1), str))
```

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtManageChildren()

NAME

XtManageChildren(), XtManageChild(), XtUnmanageChildren(), XtUnmanageChild() – manage and unmanage children

SYNOPSIS

```
typedef Widget *WidgetList

void XtManageChildren (WidgetList children, Cardinal num_children)
void XtManageChild (Widget child)
void XtUnmanageChildren (WidgetList children, Cardinal num_children)
void XtUnmanageChild (Widget child)
```

ARGUMENTS

child Specifies the child.
children Specifies a list of child widgets.
num_children
Specifies the number of children.

DESCRIPTION

The *XtManageChildren()* function performs the following:

- Issues an error if the children do not all have the same parent or if the parent is not a subclass of *compositeWidgetClass*.
- Returns immediately if the common parent is being destroyed; otherwise, for each unique child on the list, *XtManageChildren()* ignores the child if it already is managed or is being destroyed and marks it if not.
- If the parent is realized and after all children have been marked, it makes some of the newly managed children viewable:
 - Calls the *change_managed* routine of the widgets' parent.
 - Calls *XtRealizeWidget()* on each previously unmanaged child that is unrealized.
 - Maps each previously unmanaged child that has *map_when_managed* True.

Managing children is independent of the ordering of children and independent of creating and deleting children. The layout routine of the parent should consider children whose *managed* field is True and should ignore all other children. Note that some composite widgets, especially fixed boxes, call *XtManageChild()* from their *insert_child* procedure.

If the parent widget is realized, its *change_managed* procedure is called to notify it that its set of managed children has changed. The parent can reposition and resize any of its children. It moves each child as needed by calling *XtMoveWidget()*, which first updates the x and y fields and then calls *XMoveWindow()* if the widget is realized.

The *XtManageChild()* function constructs a *WidgetList()* of length one and calls *XtManageChildren()*.

The *XtUnmanageChildren()* function performs the following:

- Issues an error if the children do not all have the same parent or if the parent is not a subclass of *compositeWidgetClass*.

XtManageChildren()

- Returns immediately if the common parent is being destroyed; otherwise, for each unique child on the list, *XtUnmanageChildren()* performs the following:
 - Ignores the child if it already is unmanaged or is being destroyed and marks it if not.
 - If the child is realized, it makes it nonvisible by unmapping it.
- Calls the *change_managed* routine of the widgets' parent after all children have been marked if the parent is realized.

XtUnmanageChildren() does not destroy the children widgets. Removing widgets from a parent's managed set is often a temporary banishment, and, some time later, you may manage the children again.

The *XtUnmanageChild()* function constructs a widget list of length one and calls *XtUnmanageChildren()*.

SEE ALSO

XtMapWidget(), *XtRealizeWidget()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtMapWidget()

NAME

XtMapWidget(), XtSetMappedWhenManaged(), XtUnmapWidget() – map and unmap widgets

SYNOPSIS

```
XtMapWidget (Widget w)
void XtSetMappedWhenManaged (Widget w, Boolean map_when_managed)
XtUnmapWidget (Widget w)
```

ARGUMENTS

map_when_managed Specifies a Boolean value that indicates the new value of the *map_when_managed* field.

w Specifies the widget.

DESCRIPTION

If the widget is realized and managed and if the new value of *map_when_managed* is True, XtSetMappedWhenManaged maps the window. If the widget is realized and managed and if the new value of *map_when_managed* is False, it unmaps the window. *XtSetMappedWhenManaged()* is a convenience function that is equivalent to (but slightly faster than) calling *XtSetValues()* and setting the new value for the mappedWhenManaged resource. As an alternative to using *XtSetMappedWhenManaged()* to control mapping, a client may set *mapped_when_managed* to False and use *XtMapWidget()* and *XtUnmapWidget()* explicitly.

SEE ALSO

XtManageChildren()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtNameToWidget()

NAME

XtNameToWidget(), XtWidgetToWindow() – translating strings to widgets or widgets to windows

SYNOPSIS

```
Widget XtNameToWidget (Widget reference, String names)  
Widget XtWindowToWidget (Display *display, Window window)
```

ARGUMENTS

display Specifies the display on which the window is defined.
names Specifies the fully qualified name of the desired widget.
reference Specifies the widget from which the search is to start.
window Specify the window for which you want the widget.

DESCRIPTION

The *XtNameToWidget()* function looks for a widget whose name is the first component in the specified names and that is a pop-up child of reference (or a normal child if reference is a subclass of compositeWidgetClass). It then uses that widget as the new reference and repeats the search after deleting the first component from the specified names. If it cannot find the specified widget, *XtNameToWidget()* returns NULL.

Note that the names argument contains the name of a widget with respect to the specified reference widget and can contain more than one widget name (separated by periods) for widgets that are not direct children of the specified reference widget.

If more than one child of the reference widget matches the name, *XtNameToWidget()* can return any of the children. The Intrinsics do not require that all children of a widget have unique names. If the specified names contain more than one component and if more than one child matches the first component, *XtNameToWidget()* can return NULL if the single branch that it follows does not contain the named widget. That is, *XtNameToWidget()* does not back up and follow other matching branches of the widget tree.

The *XtWindowToWidget()* function translates the specified window and display pointer into the appropriate widget instance.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtOffset()

NAME

XtOffset(), XtNumber() – determine the byte offset or number of array elements

SYNOPSIS

Cardinal XtOffset (Type *pointer_type*, Field *field_name*)
Cardinal XtNumber (ArrayVariable *array*)

ARGUMENTS

array Specifies a fixed-size array.
field_name Specifies the name of the field for which to calculate the byte offset.
pointer_type Specifies a type that is declared as a pointer to the structure.

DESCRIPTION

The *XtOffset()* macro is usually used to determine the offset of various resource fields from the beginning of a widget and can be used at compile time in static initializations.

The *XtNumber()* macro returns the number of elements in the specified argument lists, resources lists, and other counted arrays.

SEE ALSO

XtGetResourceList(), *XtSetArg()*
X Toolkit Intrinsics – C Language Interface
Xlib – C Language X Interface

XtOwnSelection()

NAME

XtOwnSelection(), XtDisownSelection() – set selection owner

SYNOPSIS

```
Boolean XtOwnSelection (Widget w, Atom selection, Time time,  
                        XtConvertSelectionProc convert_proc,  
                        XtLoseSelectionProc lose_selection,  
                        XtSelectionDoneProc done_proc)  
  
void XtDisownSelection (Widget w, Atom selection, Time time)
```

ARGUMENTS

convert_proc

Specifies the procedure that is to be called whenever someone requests the current value of the selection.

done_proc

Specifies the procedure that is called after the requestor has received the selection or NULL if the owner is not interested in being called back.

lose_selection

Specifies the procedure that is to be called whenever the widget has lost selection ownership or NULL if the owner is not interested in being called back.

selection Specifies an atom that describes the type of the selection (for example, XA_PRIMARY, XA_SECONDARY, or XA_CLIPBOARD).

time Specifies the timestamp that indicates when the selection ownership should commence or is to be relinquished.

w Specifies the widget that wishes to become the owner or to relinquish ownership.

DESCRIPTION

The *XtOwnSelection()* function informs the Intrinsics selection mechanism that a widget believes it owns a selection. It returns True if the widget has successfully become the owner and False otherwise. The widget may fail to become the owner if some other widget has asserted ownership at a time later than this widget. Note that widgets can lose selection ownership either because someone else asserted later ownership of the selection or because the widget voluntarily gave up ownership of the selection. Also note that the *lose_selection* procedure is not called if the widget fails to obtain selection ownership in the first place.

The *XtDisownSelection()* function informs the Intrinsics selection mechanism that the specified widget is to lose ownership of the selection. If the widget does not currently own the selection either because it lost the selection or because it never had the selection to begin with, *XtDisownSelection()* does nothing.

After a widget has called *XtDisownSelection()*, its convert procedure is not called even if a request arrives later with a timestamp during the period that this widget owned the selection. However, its done procedure will be called if a conversion that started before the call to *XtDisownSelection()* finishes after the call to *XtDisownSelection()*.

SEE ALSO

XtAppGetSelectionTimeout(), *XtGetSelectionValue()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtParseAcceleratorTable()

NAME

XtParseAcceleratorTable(), XtInstallAccelerators(), XtInstallAllAccelerators() – managing accelerator tables

SYNOPSIS

```
XtAccelerators XtParseAcceleratorTable (String source)  
void XtInstallAccelerators (Widget destination, Widget source)  
void XtInstallAllAccelerators (Widget destination, Widget source)
```

ARGUMENTS

source Specifies the accelerator table to compile.
destination Specifies the widget on which the accelerators are to be installed.
source Specifies the widget or the root widget of the widget tree from which the accelerators are to come.

DESCRIPTION

The *XtParseAcceleratorTable()* function compiles the accelerator table into the opaque internal representation.

The *XtInstallAccelerators()* function installs the accelerators from *source* onto *destination* by augmenting the destination translations with the source accelerators. If the *source* *display_accelerator* method is non-NULL, *XtInstallAccelerators()* calls it with the *source* widget and a string representation of the accelerator table, which indicates that its accelerators have been installed and that it should display them appropriately. The string representation of the accelerator table is its canonical translation table representation.

The *XtInstallAllAccelerators()* function recursively descends the widget tree rooted at *source* and installs the accelerators of each widget encountered onto *destination*. A common use is to call *XtInstallAllAccelerators()* and pass the application main window as the *source*.

SEE ALSO

XtParseTranslationTable

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtParseTranslationTable()

NAME

XtParseTranslationTable(), XtAugmentTranslations(), XtOverrideTranslations(),
XtUninstallTranslations() – manage translation tables

SYNOPSIS

```
XtTranslations XtParseTranslationTable (String table)  
void XtAugmentTranslations (Widget w, XtTranslations translations)  
void XtOverrideTranslations (Widget w, XtTranslations translations)  
void XtUninstallTranslations (Widget w)
```

ARGUMENTS

table Specifies the translation table to compile.
translations Specifies the compiled translation table to merge in (must not be NULL).
w Specifies the widget into which the new translations are to be merged or removed.

DESCRIPTION

The *XtParseTranslationTable()* function compiles the translation table into the opaque internal representation of type **XtTranslations**. Note that if an empty translation table is required for any purpose, one can be obtained by calling *XtParseTranslationTable()* and passing an empty string.

The *XtAugmentTranslations()* function nondestructively merges the new translations into the existing widget translations. If the new translations contain an event or event sequence that already exists in the widget's translations, the new translation is ignored.

The *XtOverrideTranslations()* function destructively merges the new translations into the existing widget translations. If the new translations contain an event or event sequence that already exists in the widget's translations, the new translation is merged in and override the widget's translation.

To replace a widget's translations completely, use *XtSetValues()* on the XtNtranslations resource and specify a compiled translation table as the value.

The *XtUninstallTranslations()* function causes the entire translation table for widget to be removed.

SEE ALSO

XtAppAddActions(), *XtCreatePopupShell()*, *XtParseAcceleratorTable()*, *XtPopup()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtPopdown()

NAME

XtPopdown(), XtCallbackPopdown(), MenuPopdown() – unmap a pop-up

SYNOPSIS

```
void XtPopdown (Widget popup_shell)
void XtCallbackPopdown (Widget w, XtPointer client_data,
                        XtPointer call_data)
void MenuPopdown (String shell_name)
```

ARGUMENTS

call_data
Specifies the callback data, which is not used by this procedure.

client_data
Specifies a pointer to the **XtPopdownID** structure.

popup_shell
Specifies the widget shell to pop down.

shell_name
Specifies the name of the widget shell to pop down.

w
Specifies the widget.

DESCRIPTION

The *XtPopdown()* function performs the following:

- Calls *XtCheckSubclass()* to ensure *popup_shell* is a subclass of Shell.
- Checks that *popup_shell* is currently popped_up; otherwise, it generates an error.
- Unmaps *popup_shell*'s window.
- If *popup_shell*'s *grab_kind* is either XtGrabNonexclusive or XtGrabExclusive, it calls *XtRemoveGrab()*.
- Sets pop-up shell's *popped_up* field to False.
- Calls the callback procedures on the shell's *popdown_callback* list.

The *XtCallbackPopdown()* function casts the client data parameter to an **XtPopdownID** pointer:

```
typedef struct {
    Widget shell_widget;
    Widget enable_widget;
} XtPopdownIDRec, *XtPopdownID;
```

The *shell_widget* is the pop-up shell to pop down, and the *enable_widget* is the widget that was used to pop it up.

XtCallbackPopdown() calls *XtPopdown()* with the specified *shell_widget* and then calls *XtSetSensitive()* to resensitize the *enable_widget*.

If a shell name is not given, *MenuPopdown()* calls *XtPopdown()* with the widget for which the translation is specified. If a *shell_name* is specified in the translation table, *MenuPopdown()* tries to find the shell by looking up the widget tree starting at the parent of the widget in which it is invoked. If it finds a shell with the specified name in the pop-up children of that parent, it pops down the shell; otherwise, it moves up the parent chain as needed. If *MenuPopdown()* gets to the application top-level shell widget and

XtPopdown()

cannot find a matching shell, it generates an error.

SEE ALSO

XtCreatePopupShell(), *XtPopup()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtPopup()

NAME

XtPopup(), XtCallbackNone(), XtCallbackNonexclusive(), XtCallbackExclusive(),
MenuPopup() – map a pop-up

SYNOPSIS

```
void XtPopup (Widget popup_shell, XtGrabKind grab_kind)
void XtCallbackNone (Widget w, XtPointer client_data,
                    XtPointer call_data)
void XtCallbackNonexclusive (Widget w, XtPointer client_data,
                           XtPointer call_data)
void XtCallbackExclusive (Widget w, XtPointer client_data,
                        XtPointer call_data)
void MenuPopup (String shell_name)
```

ARGUMENTS

call_data
Specifies the callback data, which is not used by this procedure.

client_data
Specifies the pop-up shell.

grab_kind
Specifies the way in which user events should be constrained.

popup_shell
Specifies the widget shell.

w
Specifies the widget.

DESCRIPTION

The *XtPopup()* function performs the following:

- Calls *XtCheckSubclass()* to ensure *popup_shell* is a subclass of Shell.
- Generates an error if the shell's *popped_up* field is already True.
- Calls the callback procedures on the shell's *popup_callback* list.
- Sets the shell *popped_up* field to True, the shell *spring_loaded* field to False, and the shell *grab_kind* field from *grab_kind*.
- If the shell's *create_popup_child* field is non-NULL, *XtPopup()* calls it with *popup_shell* as the parameter.
- If *grab_kind* is either *XtGrabNonexclusive* or *XtGrabExclusive*, it calls:

```
XtAddGrab(popup_shell, (grab_kind == XtGrabExclusive), False)
```
- Calls *XtRealizeWidget()* with *popup_shell* specified.
- Calls *XMapWindow()* with *popup_shell* specified.

The *XtCallbackNone()*, *XtCallbackNonexclusive()*, and *XtCallbackExclusive()* functions call *XtPopup()* with the shell specified by the client data argument and *grab_kind* set as the name specifies. *XtCallbackNone()*, *XtCallbackNonexclusive()*, and *XtCallbackExclusive()* specify *XtGrabNone()*, *XtGrabNonexclusive()*, and *XtGrabExclusive()*, respectively. Each function then sets the widget that executed the callback list to be insensitive by using *XtSetSensitive()*. Using these functions in callbacks is not required. In particular, an application must provide customized code for callbacks that create pop-up shells dynamically or that must do more than desensitizing the button.

XtPopup()

MenuPopup() is known to the translation manager, which must perform special actions for spring-loaded pop-ups. Calls to *MenuPopup()* in a translation specification are mapped into calls to a nonexported action procedure, and the translation manager fills in parameters based on the event specified on the left-hand side of a translation.

If *MenuPopup()* is invoked on *ButtonPress* (possibly with modifiers), the translation manager pops up the shell with *grab_kind* set to *XtGrabExclusive* and *spring_loaded* set to *True*. If *MenuPopup()* is invoked on *EnterWindow* (possibly with modifiers), the translation manager pops up the shell with *grab_kind* set to *XtGrabNonexclusive* and *spring_loaded* set to *False*. Otherwise, the translation manager generates an error. When the widget is popped up, the following actions occur:

- Calls *XtCheckSubclass()* to ensure *popup_shell* is a subclass of *Shell*.
- Generates an error if the shell's *popped_up* field is already *True*.
- Calls the callback procedures on the shell's *popup_callback* list.
- Sets the shell *popped_up* field to *True* and the shell *grab_kind* and *spring_loaded* fields appropriately.
- If the shell's *create_popup_child* field is non-NULL, it is called with *popup_shell* as the parameter.
- Calls:
 - `XtAddGrab(popup_shell, (grab_kind == XtGrabExclusive), spring_loaded)`
- Calls *XtRealizeWidget()* with *popup_shell* specified.
- Calls *XMapWindow()* with *popup_shell* specified.

(Note that these actions are the same as those for *XtPopup()*.) *MenuPopup()* tries to find the shell by searching the widget tree starting at the parent of the widget in which it is invoked. If it finds a shell with the specified name in the pop-up children of that parent, it pops up the shell with the appropriate parameters. Otherwise, it moves up the parent chain as needed. If *MenuPopup()* gets to the application widget and cannot find a matching shell, it generates an error.

SEE ALSO

XtCreatePopupShell(), *XtPopdown()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtQueryGeometry()

NAME

XtQueryGeometry() – query the preferred geometry of a child widget

SYNOPSIS

```
XtGeometryResult XtQueryGeometry (Widget w,  
                                   XtWidgetGeometry *intended,  
                                   XtWidgetGeometry *preferred_return)
```

ARGUMENTS

intended Specifies any changes the parent plans to make to the child's geometry or NULL.

preferred_return Returns the child widget's preferred geometry.

w Specifies the widget.

DESCRIPTION

To discover a child's preferred geometry, the child's parent sets any changes that it intends to make to the child's geometry in the corresponding fields of the intended structure, sets the corresponding bits in *intended.request_mode*, and calls I.Fn XtQueryGeometry .

XtQueryGeometry clears all bits in the *preferred_return->request_mode* and checks the *query_geometry* field of the specified widget's class record. If *query_geometry* is not NULL, *XtQueryGeometry()* calls the *query_geometry* procedure and passes as arguments the specified widget, *intended*, and *preferred_return* structures. If the intended argument is NULL, *XtQueryGeometry()* replaces it with a pointer to an **XtWidgetGeometry** structure with *request_mode=0* before calling *query_geometry*.

SEE ALSO

XtConfigureWidget(), *XtMakeGeometryRequest()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtRealizeWidget()

NAME

XtRealizeWidget(), XtIsRealized(), XtUnrealizeWidget() – realize and unrealize widgets

SYNOPSIS

```
void XtRealizeWidget (Widget w)
Boolean XtIsRealized (Widget w)
void XtUnrealizeWidget (Widget w)
```

ARGUMENTS

w Specifies the widget.

DESCRIPTION

If the widget is already realized, *XtRealizeWidget()* simply returns. Otherwise, it performs the following:

- Binds all action names in the widget's translation table to procedures (see Section 10.1.2).
- Makes a post-order traversal of the widget tree rooted at the specified widget and calls the *change_managed* procedure of each composite widget that has one or more managed children.
- Constructs an **XSetWindowAttributes** structure filled in with information derived from the *Core* widget fields and calls the *realize* procedure for the widget, which adds any widget-specific attributes and creates the X window.
- If the widget is not a subclass of *compositeWidgetClass*, *XtRealizeWidget()* returns; otherwise, it continues and performs the following:
 - Descends recursively to each of the widget's managed children and calls the *realize* procedures. Primitive widgets that instantiate children are responsible for realizing those children themselves.
 - Maps all of the managed children windows that have *mapped_when_managed* True. (If a widget is managed but *mapped_when_managed* is False, the widget is allocated visual space but is not displayed. Some people seem to like this to indicate certain states.)

If the widget is a top-level shell widget (that is, it has no parent), and *mapped_when_managed* is True, *XtRealizeWidget()* maps the widget window.

The *XtIsRealized()* function returns True if the widget has been realized, that is, if the widget has a nonzero X window ID.

Some widget procedures (for example, *set_values*) might wish to operate differently after the widget has been realized.

The *XtUnrealizeWidget()* function destroys the windows of an existing widget and all of its children (recursively down the widget tree). To recreate the windows at a later time, call *XtRealizeWidget()* again. If the widget was managed, it will be unmanaged automatically before its window is freed.

SEE ALSO

XtManageChildren()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtSetArg()

NAME

XtSetArg(), XtMergeArgLists() – set and merge ArgLists

SYNOPSIS

```
XtSetArg (Arg arg, String name, XtArgVal value)
ArgList XtMergeArgLists (ArgList args1, Cardinal num_args1,
                          ArgList args2, Cardinal num_args2)
```

ARGUMENTS

arg Specifies the name-value pair to set.
args1 Specifies the first **ArgList**.
args2 Specifies the second **ArgList**.
num_args1 Specifies the number of arguments in the first argument list.
num_args2 Specifies the number of arguments in the second argument list.
name Specifies the name of the resource.
value Specifies the value of the resource if it will fit in an **XtArgVal** or the address.

DESCRIPTION

The *XtSetArg()* function is usually used in a highly stylized manner to minimize the probability of making a mistake; for example:

```
Arg args[20];
int n;

n = 0;
XtSetArg(args[n], XtNheight, 100); n++;
XtSetArg(args[n], XtNwidth, 200); n++;
XtSetValues(widget, args, n);
```

Alternatively, an application can statically declare the argument list and use *XtNumber()*:

```
static Argv args[] = {
    {XtNheight, (XtArgVal) 100},
    {XtNwidth, (XtArgVal) 200},
};
XtSetValues(widget, args, XtNumber(args));
```

Note that you should not use auto-increment or auto-decrement within the first argument to *XtSetArg()*. *XtSetArg()* can be implemented as a macro that dereferences the first argument twice.

The *XtMergeArgLists()* function allocates enough storage to hold the combined **ArgList** structures and copies them into it. Note that it does not check for duplicate entries. When it is no longer needed, free the returned storage by using **XtFree**.

SEE ALSO

XtOffset()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtSetKeyboardFocus()

NAME

XtSetKeyboardFocus() – focus events on a child widget

SYNOPSIS

XtSetKeyboardFocus (Widget *subtree*, Widget *descendant*)

ARGUMENTS

descendant

Specifies either the widget in the subtree structure which is to receive the keyboard event, or None. Note that it is not an error to specify None when no input focus was previously set.

w

Specifies the widget for which the keyboard focus is to be set.

DESCRIPTION

If a future KeyPress or KeyRelease event occurs within the specified *subtree*, *XtSetKeyboardFocus()* causes *XtDispatchEvent()* to remap and send the event to the specified descendant widget.

When there is no modal cascade, keyboard events can occur within a widget W in one of three ways:

- W has the X input focus.
- W has the keyboard focus of one of its ancestors, and the event occurs within the ancestor or one of the ancestor's descendants.
- No ancestor of W has a descendant within the keyboard focus, and the pointer is within W.

When there is a modal cascade, a widget W receives keyboard events if an ancestor of W is in the active subset of the modal cascade and one or more of the previous conditions is True.

When *subtree* or one of its descendants acquires the X input focus or the pointer moves into the *subtree* such that keyboard events would now be delivered to *subtree*, a FocusIn event is generated for the descendant if FocusNotify events have been selected by the descendant. Similarly, when W loses the X input focus or the keyboard focus for one of its ancestors, a FocusOut event is generated for descendant if FocusNotify events have been selected by the descendant.

SEE ALSO

XtCallAcceptFocus()

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtSetKeyTranslator()

NAME

XtSetKeyTranslator(), XtTranslateKeycode(), XtRegisterCaseConverter(),
XtConvertCase() – convert KeySym to KeyCodes

SYNOPSIS

```
void XtSetKeyTranslator (Display *display, XtKeyProc proc)
void XtTranslateKeycode (Display *display, KeyCode keycode,
                        Modifiers modifiers,
                        Modifiers *modifiers_return,
                        KeySym *keysym_return)
void XtRegisterCaseConverter (Display *display, XtCaseProc proc,
                             KeySym start, KeySym stop)
void XtConvertCase (Display *display, KeySym keysym,
                   KeySym *lower_return, KeySym *upper_return)
```

ARGUMENTS

display Specifies the display.
keycode Specifies the KeyCode to translate.
keysym Specifies the KeySym to convert.
keysym_return
Returns the resulting KeySym.
lower_return
Returns the lowercase equivalent of the KeySym.
upper_return
Returns the uppercase equivalent of the KeySym.
modifiers
Specifies the modifiers to the KeyCode.
modifiers_return
Returns a mask that indicates the modifiers actually used to generate the KeySym.
proc Specifies the procedure that is to perform key translations or conversions.
start Specifies the first KeySym for which this converter is valid.
stop Specifies the last KeySym for which this converter is valid.

DESCRIPTION

The *XtSetKeyTranslator()* function sets the specified procedure as the current key translator. The default translator is *XtTranslateKey()*, an *XtKeyProc()* that uses Shift and Lock modifiers with the interpretations defined by the core protocol. It is provided so that new translators can call it to get default KeyCode-to-Keysym translations and so that the default translator can be reinstalled.

The *XtTranslateKeycode()* function passes the specified arguments directly to the currently registered KeyCode to Keysym translator.

The *XtRegisterCaseConverter()* registers the specified case converter. The start and stop arguments provide the inclusive range of Keysyms for which this converter is to be called. The new converter overrides any previous converters for Keysyms in that range. No interface exists to remove converters; you need to register an identity converter. When a new converter is registered, the Intrinsics refreshes the keyboard state if necessary. The default converter understands case conversion for all Keysyms defined in the core protocol.

XtSetKeyTranslator()

The *XtConvertCase()* function calls the appropriate converter and returns the results. A user-supplied *XtKeyProc()* may need to use this function.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtSetSensitive()

NAME

XtSetSensitive(), XtIsSensitive() – set and check a widget’s sensitivity state

SYNOPSIS

```
void XtSetSensitive (Widget w, Boolean sensitive)
Boolean XtIsSensitive (Widget w)
```

ARGUMENTS

sensitive Specifies a Boolean value that indicates whether the widget should receive keyboard and pointer events.
w Specifies the widget.

DESCRIPTION

The *XtSetSensitive()* function first calls *XtSetValues()* on the current widget with an argument list specifying that the *sensitive* field should change to the new value. It then recursively propagates the new value down the managed children tree by calling *XtSetValues()* on each child to set the *ancestor_sensitive* to the new value if the new values for *sensitive* and the child’s *ancestor_sensitive* are not the same.

XtSetSensitive() calls *XtSetValues()* to change *sensitive* and *ancestor_sensitive*. Therefore, when one of these changes, the widget’s *set_values* procedure should take whatever display actions are needed (for example, greying out or stippling the widget).

XtSetSensitive() maintains the invariant that if parent has either *sensitive* or *ancestor_sensitive* False, then all children have *ancestor_sensitive* False.

The *XtIsSensitive()* function returns True or False to indicate whether or not user input events are being dispatched. If both *core.sensitive* and *core.ancestor_sensitive* are True, *XtIsSensitive()* returns True; otherwise, it returns False.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtSetValues()

NAME

XtSetValues(), XtSetSubvalues(), XtGetValues(), XtGetSubvalues() – obtain and set widget resources

SYNOPSIS

```
void XtSetValues (Widget w, ArgList args, Cardinal num_args)
void XtSetSubvalues (XtPointer base, XtResourceList resources,
                    Cardinal num_resources, ArgList args,
                    Cardinal num_args)
void XtGetValues (Widget w, ArgList args, Cardinal num_args)
void XtGetSubvalues (XtPointer base, XtResourceList resources,
                    Cardinal num_resources, ArgList args,
                    Cardinal num_args)
```

ARGUMENTS

<i>args</i>	Specifies the argument list of name / address pairs that contain the resource name and either the address into which the resource value is to be stored or their new values.
<i>base</i>	Specifies the base address of the subpart data structure where the resources should be retrieved or written.
<i>num_args</i>	Specifies the number of arguments in the argument list.
<i>resources</i>	Specifies the nonwidget resource list or values.
<i>num_resources</i>	Specifies the number of resources in the resource list.
<i>w</i>	Specifies the widget.

DESCRIPTION

The *XtSetValues()* function starts with the resources specified for the **Core** widget fields and proceeds down the subclass chain to the widget. At each stage, it writes the new value (if specified by one of the arguments) or the existing value (if no new value is specified) to a new widget data record. *XtSetValues()* then calls the *set_values* procedures for the widget in superclass-to-subclass order. If the widget has any non-NULL *set_values_hook* fields, these are called immediately after the corresponding *set_values* procedure. This procedure permits subclasses to set nonwidget data for *XtSetValues()*.

If the widget's parent is a subclass of *constraintWidgetClass*, *XtSetValues()* also updates the widget's constraints. It starts with the constraint resources specified for *constraintWidgetClass()* and proceeds down the subclass chain to the parent's class. At each stage, it writes the new value or the existing value to a new constraint record. It then calls the constraint *set_values* procedures from *constraintWidgetClass* down to the parent's class. The constraint *set_values* procedures are called with widget arguments, as for all *set_values* procedures, not just the constraint record arguments, so that they can make adjustments to the desired values based on full information about the widget.

XtSetValues() determines if a geometry request is needed by comparing the current widget to the new widget. If any geometry changes are required, it makes the request, and the geometry manager returns *XtGeometryYes*, *XtGeometryAlmost*, or *XtGeometryNo*. If *XtGeometryYes*, *XtSetValues()* calls the widget's *resize* procedure. If *XtGeometryNo*, *XtSetValues()* resets the geometry fields to their original values. If

XtSetValues()

XtGeometryAlmost, *XtSetValues()* calls the *set_values_almost* procedure, which determines what should be done and writes new values for the geometry fields into the new widget. *XtSetValues()* then repeats this process, deciding once more whether the geometry manager should be called.

Finally, if any of the *set_values* procedures returned True, *XtSetValues* causes the widget's expose procedure to be invoked by calling the Xlib *XClearArea()* function on the widget's window.

The *XtSetSubvalues()* function stores resources into the structure identified by base.

The *XtGetValues()* function starts with the resources specified for the core widget fields and proceeds down the subclass chain to the widget. The value field of a passed argument list should contain the address into which to store the corresponding resource value. It is the caller's responsibility to allocate and deallocate this storage according to the size of the resource representation type used within the widget.

If the widget's parent is a subclass of *constraintWidgetClass*, *XtGetValues()* then fetches the values for any constraint resources requested. It starts with the constraint resources specified for *constraintWidgetClass()* and proceeds down to the subclass chain to the parent's constraint resources. If the argument list contains a resource name that is not found in any of the resource lists searched, the value at the corresponding address is not modified. Finally, if the *get_values_hook* procedures are non-NULL, they are called in superclass-to-subclass order after all the resource values have been fetched by *XtGetValues()*. This permits a subclass to provide nonwidget resource data to *XtGetValues()*.

The *XtGetSubvalues()* function obtains resource values from the structure identified by base.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtStringConversionWarning()

NAME

XtStringConversionWarning() – issue a conversion warning message

SYNOPSIS

```
void XtStringConversionWarning (String src, dst_type)
```

ARGUMENTS

src Specifies the string that could not be converted.

dst_type Specifies the name of the type to which the string could not be converted.

DESCRIPTION

The *XtStringConversionWarning()* function issues a warning message with name “conversionError”, type “string”, class “XtToolkitError, and the default message string “Cannot convert “*src*” to type *dst_type*”.

SEE ALSO

XtAppAddConverter(), *XtAppErrorMsg()*, *XtConvert()*

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface

XtTranslateCoordinates()

NAME

XtTranslateCoordinates() – translate widget coordinates

SYNOPSIS

```
void XtTranslateCoords (Widget w, Position x, Position y,  
                        Position *rootx_return, *rooty_return)
```

ARGUMENTS

rootx_return

rooty_return

Returns the root-relative x and y coordinates.

x

y

Specify the widget-relative x and y coordinates.

w

Specifies the widget.

DESCRIPTION

While *XtTranslateCoords()* is similar to the Xlib *XTranslateCoordinates()* function, it does not generate a server request because all the required information already is in the widget's data structures.

SEE ALSO

X Toolkit Intrinsics – C Language Interface

Xlib – C Language X Interface



Chapter 1

X Extensions Library

NAME

XShapeQueryExtension(), XShapeQueryVersion(), XShapeCombineRegion(),
 XShapeCombineRectangles(), XShapeCombineMask(), XShapeCombineShape(),
 XShapeOffsetShape(), XShapeQueryExtents(), XShapeSelectInput(),
 XShapeInputSelected(), XShapeGetRectangles() – X nonrectangular shape functions

SYNOPSIS

```
#include <X11/extensions/shape.h>

Bool XShapeQueryExtension (Display *dpy, int *event_basep,
                           int *error_basep)
Status XShapeQueryVersion (Display *dpy, int *major_versionp,
                           int *minor_versionp)
void XShapeCombineRegion (Display *dpy, Window dest,
                          int destKind, int xOff, int yOff,
                          struct _XRegion *r, int op)
void XShapeCombineRectangles (Display *dpy, XID dest,
                              int destKind, int xOff, int yOff,
                              XRectangle *rects, int n_rects,
                              int op, int ordering)
void XShapeCombineMask (Display *dpy, XID dest, int destKind,
                       int xOff, int yOff, Pixmap src,
                       int op)
void XShapeCombineShape (Display *dpy, XID dest, int destKind,
                        int xOff, int yOff, Pixmap src,
                        int srcKind, int op)
void XShapeOffsetShape (Display *dpy, XID dest, int destKind,
                       int xOff, int yOff)
Status XShapeQueryExtents (Display *dpy, Window window, int *bShaped,
                           int *xbs, int *ybs,
                           unsigned int *wbs, unsigned int *hbs,
                           int *cShaped, int *xcs, int *ycs,
                           unsigned int *wcs, unsigned int *hcs)
void XShapeSelectInput (Display *dpy, Window window,
                       unsigned long mask)
unsigned long XShapeInputSelected (Display *dpy, Window window)
XRectangle *XShapeGetRectangles (Display *dpy, Window window,
                                 int kind, int *count,
                                 int *ordering)
```

STRUCTURES

typedef struct {	
int type;	of event
unsigned long serial;	# of last request processed by server
Bool send_event;	true if this came from a SendEvent request
Display *display;	Display the event was read from
Window window;	window of event
int kind;	ShapeBounding or ShapeClip
int x, y;	extents of new region
unsigned width, height;	
Time time;	server timestamp when region changed

```
        Bool shaped;                true if the region exists
    } XShapeEvent;
```

DESCRIPTION

The *X11 Nonrectangular Window Shape Extension* adds nonrectangular windows to the X Window System.

PREDEFINED VALUES

Operations:

- ShapeSet
- ShapeUnion
- ShapeIntersect
- ShapeSubtract
- ShapeInvert

Shape Kinds:

- ShapeBounding
- ShapeClip

Event defines:

- ShapeNotifyMask
- ShapeNotify

BUGS

This manual page needs a lot more work.

SEE ALSO

X11 Nonrectangular Window Shape Extension

NAME

XmbufQueryExtension(), XmbufGetVersion(), XmbufCreateBuffers(),
 XmbufDestroyBuffers(), XmbufDisplayBuffers(), XmbufGetWindowAttributes(),
 XmbufChangeWindowAttributes(), XmbufGetBufferAttributes(),
 XmbufChangeBufferAttributes(), XmbufGetScreenInfo(), XmbufCreateStereoWindow()
 – X multibuffering functions

SYNOPSIS

```
#include <X11/extensions/multibuf.h>

Bool XmbufQueryExtension (Display *dpy, Display *dpy,
                          int *event_base_return,
                          int *error_base_return)

Status XmbufGetVersion (Display *dpy, int *major_version_return,
                       int *minor_version_return)

int XmbufCreateBuffers (Display *dpy, Window window, int count,
                       int update_action, int update_hint,
                       Multibuffer *buffers_update)

void XmbufDestroyBuffers (Display *dpy, Window window)

void XmbufDisplayBuffers (Display *dpy, int count,
                          Multibuffer *buffers, int min_delay,
                          int max_delay)

Status XmbufGetWindowAttributes (Display *dpy, Window window,
                                XmbufWindowAttributes *attributes)

void XmbufChangeWindowAttributes (Display *dpy, Window window,
                                unsigned long valuemask,
                                XmbufSetWindowAttributes *attributes)

Status XmbufGetBufferAttributes (Display *dpy, Multibuffer buffer,
                                XmbufBufferAttributes *attributes)

void XmbufChangeBufferAttributes (Display *dpy, Multibuffer buffer,
                                unsigned long valuemask,
                                XmbufSetBufferAttributes *attributes)

Status XmbufGetScreenInfo (Display *dpy, Drawable drawable,
                           int *nmono_return, XmbufBufferInfo **mono_info_return,
                           int *nstereo_return,
                           XmbufBufferInfo **stereo_info_return)

Window XmbufCreateStereoWindow (Display *dpy, Window parent,
                                int x, int y,
                                unsigned int width, unsigned int height,
                                unsigned int border_width, int depth,
                                unsigned int class,
                                Visual *visual,
                                unsigned long valuemask,
                                XSetWindowAttributes *attributes,
                                Multibuffer *left_return,
                                Multibuffer *right_return)
```

STRUCTURES

Events:

```

typedef struct {
    int type;                of event
    unsigned long serial;    # of last request processed by server
    int send_event;          true if this came from a SendEvent request
    Display *display;        Display the event was read from
    Multibuffer buffer;      buffer of event
    int state;               see Clobbered constants above
} XmbufClobberNotifyEvent;

typedef struct {
    int type; of event
    unsigned long serial; # of last request processed by server
    int send_event; true if this came from a SendEvent request
    Display *display; Display the event was read from
    Multibuffer buffer; buffer of event
} XmbufUpdateNotifyEvent;

```

Per-window attributes that can be got:

```

typedef struct {
    int displayed_index;    which buffer is being displayed
    int update_action;      Undefined, Background, Untouched, Copied
    int update_hint;        Frequent, Intermittent, Static
    int window_mode;        Mono, Stereo
    int nbuffers;           Number of buffers
    Multibuffer *buffers;   Buffers
} XmbufWindowAttributes;

```

Per-window attributes that can be set:

```

typedef struct {
    int update_hint;        Frequent, Intermittent, Static
} XmbufSetWindowAttributes;

```

Per-buffer attributes that can be got:

```

typedef struct {
    Window window;    which window this belongs to
    unsigned long event_mask;    events that have been selected
    int buffer_index;    which buffer is this
    int side; Mono, Left, Right
} XmbufBufferAttributes;

```

Per-buffer attributes that can be set:

```

typedef struct {
    unsigned long event_mask;    events that have been selected
} XmbufSetBufferAttributes;

```

Per-screen buffer info (there will be lists of them):

```

typedef struct {
    VisualID visualid;    visual usable at this depth
    int max_buffers;    most buffers for this visual
    int depth;    depth of buffers to be created
} XmbufBufferInfo;

```

DESCRIPTION

The application programming library for the *X11 Double-Buffering, Multi-Buffering, and Stereo Extension* contains the interfaces described below. With the exception of *XmbufQueryExtension()*, if any of these routines are called with a display that does not support the extension, the *ExtensionErrorHandler* (which can be set with *XSetExtensionErrorHandler()* and functions the same way as *XSetErrorHandler()*) will be called and the function will then return.

XmbufQueryExtension() returns True if the multibuffering / stereo extension is available on the given display. If the extension exists, the value of the first event code (which should be added to the event type constants *MultibufferClobberNotify* and *MultibufferUpdateNotify* to get the actual values) is stored into *event_base_return* and the value of the first error code (which should be added to the error type constant *MultibufferBadBuffer* to get the actual value) is stored into *error_base_return*.

XmbufGetVersion() gets the major and minor version numbers of the extension. The return value is zero if an error occurs or non-zero if no error happens.

XmbufCreateBuffers() requests that "count" buffers be created with the given *update_action* and *update_hint* and be associated with the indicated window. The number of buffers created is returned (zero if an error occurred) and *buffers_update* is filled in with that many **Multibuffer** identifiers.

XmbufDestroyBuffers() destroys the buffers associated with the given window.

XmbufDisplayBuffers() displays the indicated buffers their appropriate windows within *max_delay* milliseconds after *min_delay* milliseconds have passed. No two buffers may be associated with the same window or else a *Matc* error is generated.

XmbufGetWindowAttributes() gets the multibuffering attributes that apply to all buffers associated with the given window. The list of buffers returns may be freed with *XFree()*. Returns non-zero on success and zero if an error occurs.

XmbufChangeWindowAttributes() sets the multibuffering attributes that apply to all buffers associated with the given window. This is currently limited to the *update_hint*.

XmbufGetBufferAttributes() gets the attributes for the indicated buffer. Returns non-zero on success and zero if an error occurs.

XmbufChangeBufferAttributes() sets the attributes for the indicated buffer. This is currently limited to the *event_mask*.

XmbufGetScreenInfo() gets the parameters controlling how mono and stereo windows may be created on the screen of the given drawable. The numbers of sets of visual and depths are returned in *nmono_return* and *nstereo_return*. If *nmono_return* is greater than zero, then *mono_info_return* is set to the address of an array of **XmbufBufferInfo** structures describing the various visuals and depths that may be used. Otherwise, *mono_info_return* is set to NULL. Similarly, *stereo_info_return* is set according to *nstereo_return*. The storage returned in *mono_info_return* and *stereo_info_return* may be released by **XFree**. If no errors are encountered, non-zero will be returned.

XmbufCreateStereoWindow creates a stereo window in the same way that **XCreateWindow** creates a mono window. The buffer ids for the left and right buffers are returned in *left_return* and *right_return*, respectively. If an extension error handler that returns is installed, None will be returned if the extension is not available on this display.

PREDEFINED VALUES

Update_action field:

MultibufferUpdateActionUndefined
MultibufferUpdateActionBackground
MultibufferUpdateActionUntouched
MultibufferUpdateActionCopied

Update_hint field:

MultibufferUpdateHintFrequent
MultibufferUpdateHintIntermittent
MultibufferUpdateHintStatic

Valuemask fields:

MultibufferWindowUpdateHint
MultibufferBufferEventMask

Mono vs. stereo and left vs. right:

MultibufferModeMono
MultibufferModeStereo
MultibufferSideMono
MultibufferSideLeft
MultibufferSideRight

Clobber state:

MultibufferUnclobbered
MultibufferPartiallyClobbered
MultibufferFullyClobbered

Event stuff:

MultibufferClobberNotifyMask
MultibufferUpdateNotifyMask
MultibufferClobberNotify
MultibufferUpdateNotify
MultibufferNumberEvents
MultibufferBadBuffer
MultibufferNumberErrors

BUGS

This manual page needs more work.

SEE ALSO

Extending X for Double Buffering, Multi-Buffering, and Stereo