HP Xlib Extensions

HEWLETT
PACKARD

HP Part No. B1171-90078
Printed in U.S.A. January 1995

Edition 2
DRAFT 4/7/98 12:51

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

UNIX is a registered trademark of UNIX Software Laboratories, Inc. in the USA and other countries.

Intellifont is a registered trademark of Agfa Corporation. CG Century Schoolbook and CG Times, based on Times New Roman under license from The Monotype Corporation plc, are products of the Agfa Corporation.

Printing History

The manual printing date and part number indicate its current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The manual part number will change when extensive changes are made.

Manual updates may be issued between editions to correct errors or document product changes. To ensure that you receive these updates or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

July 1992	Edition 1
January 1995	Edition 2

Hewlett-Packard Company
Workstation Systems Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330

Contents

1

Introduction to Xlib HP Extensions

To provide better integration with existing products and peripherals available with HP 9000 computers, a number of extensions have been added to Hewlett-Packard's Xlib software. These extensions add to the existing X standard, creating a superset of functionality:

- The XLFD syntax has been extended to allow specifications for increasing or decreasing boldness, changing the slant, mirroring, rotating, and increasing the horizontal size. These extensions apply to scalable typefaces as well as to bitmap fonts. Refer to chapter 2.
- Multiple error-handling routines for a single process are allowed. Refer to chapter 3.
- Client programs can disable or enable the key sequence used to reset the X server. Refer to chapter 4.
- Two sets of input extensions are included: the "standard" input extensions and HP input extensions. The standard input extensions should be used if possible. Refer to chapter 5 for the standard input device extensions and chapter 6 for the HP input device extensions.
- HP support of internationalized text is described in chapter 7.
- Man pages for the standard extensions and the HP proprietary extensions provide more information about these extensions.

Note

There is some overlap in functionality between the HP proprietory extension protocol requests and the X standard (X11 R5) input extension protocol requests. This is because Hewlett-Packard introduced some capabilities before a standard existed. Hewlett-Packard continues to provide the proprietary extensions for backward compatibility. If either meets your needs, use the X standard capabilities, not the HP extensions. Hewlett-Packard may at some time in future releases discontinue the proprietary extensions.

The HP extensions will work among all networked HP 9000 computers, but may not work with other systems on the same network that are from other vendors or are running earlier versions of X11.

What This Manual Covers

This manual covers the extensions to Xlib (Release 5, Version 11) that were created by Hewlett-Packard. Also covered are the X standard extended input functions that supersede the HP extended input functions.

Note ☝ This manual does *not* document X standard Xlib, which is covered by the *Xlib Reference Manual*.

The Multi-Buffered Extension (MBX) is covered in *PEXLIB Programming Guide* by O'Reilly & Associates, Inc (HP Part Number B3176-90003).

Conventions Used in This Manual

This document uses the following conventions:

- Global symbols in this manual are printed in `this special font`. These can be either function names, symbols defined in include files, or structure names. Arguments are printed in *italics*.
- Each function is introduced by a general discussion that distinguishes it from other functions. The function declaration itself follows, and each argument is specifically explained. General discussion of the function, if any is required, follows the arguments. Where applicable, the last paragraph of the explanation lists the possible Xlib error codes that the function can generate.
- To eliminate any ambiguity between those arguments that you pass and those that a function returns to you, the explanations for all arguments that you pass start with the word *specifies* or, in the case of multiple arguments, the word *specify*. The explanations for all arguments that are returned to you start with the word *returns* or, in the case of multiple arguments, the word *return*. The explanations for all arguments that you can pass and are returned start with the words *specifies and returns*.
- Any pointer to a structure that is used to return a value is designated as such by the *_return* suffix as part of its name. All other pointers passed to these functions are used for reading only.
- Xlib defines the Boolean values of `True` and `False`.

2

Font Extensions

X11 supports scalable typefaces in addition to bitmapped fonts. The use of the XLFD (X Logical Font Description) for specifying either kind is covered in “Using Fonts” in the *Using the X Window System* manual.

Hewlett-Packard has extended the XLFD conventions to provide for additional font capabilities. This chapter discusses how the XLFD name can be used to modify boldness, slant, and width, for either bitmapped fonts or scalable typefaces using the HP extensions.

Changing Font Boldness

The user can specify that the font be bolder (darker) or less bold (lighter) than the normal for that typeface. The syntax for this extension is:

`Weight_Name`±*horiz_value*±*vert_value*

where:

horiz_value The increase(+) or decrease(_) in boldness in the horizontal direction, specified in $\frac{1}{100}$ of a percent.

vert_value The increase(+) or decrease(_) in boldness in the vertical direction, specified in $\frac{1}{100}$ of a percent.

If only one delta and value are supplied, they apply to both directions.

The ability to change font boldness is currently supported only for Agfa Intellifont scalable fonts. This enhancement is ignored for Type 1 fonts and scaled bitmaps.

Changing Font Slant

The user can increase or decrease the slant of the font. The syntax for this extension is:

`Slant`±*value*

where:

value The angle in $\frac{1}{64}$ degree. Counterclockwise angles are indicated by +, clockwise angles by _ . The maximum slant is ±75°.

Mirroring or Rotating

The user can specify that the font be rotated or mirrored. The syntax for this extension is:

`AddStyleName+Mx+My`±*angle*

where:

+Mx Mirrors the font horizontally.

+My Mirrors the font vertically.

angle Rotates the font from normal. The angle is measured in $\frac{1}{64}$ degree. Counterclockwise angles are indicated by +, clockwise angles by _.

Changing Horizontal Size

The horizontal size of a font can be changed to make it wider or narrower than normal for that font. Either PixelSize or PointSize can be used for this purpose. But using both PixelSize and PointSize is likely to result in an error because of a conflict between the two specifications.

The user can expand the horizontal size (pixel width) to make a font wider or narrower than normal for that font. The syntax for this extension is:

`PixelSize+`*pixelwidth*

where

pixelwidth The design width of the font, in pixels. If *pixelwidth* is not specified, the design width is assumed to be the same as `PixelSize`.

The user can expand the horizontal size (set size) to make a font wider or narrower than normal for that font. The syntax for this extension is:

`PointSize+`*setsize*

where:

setsize The set size in decipoints. If *setsize* is not specified, the set size is assumed to be the same as `PointSize`.

If neither `PixelSize` nor `PointSize` is specified, 12-point is used. If both are specified and they conflict, an error is returned. Use *either* `PixelSize` or `PointSize`, but not both.

Specifying a Character Subset

The user can specify that only certain characters from the character set be used in creating a font from the scalable typeface. The syntax for this extension is:

`CharSetEncoding=`*value*`,`*value*...

where:

value The character number, or range of character numbers, to be included in the font. A range of numbers is indicated by two numbers separated by a colon(:).

If an application requests a character not in the subset, then:

- A space will be substituted for that character *if* space is in the subset.
- The first character of the subset will be substituted if space is *not* in the subset.

Note

Subsetting will not work when specifying a Motif fontlist via resources.

3

Support for Multiple Error Handlers

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

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

typedef int (*PFI) ();

PFI XHPSetErrorHandler(display, routine)
        Display            *display;
        int                (*routine) ();

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

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

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

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

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

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

4

Locking an X Display

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

Disabling the Reset Key Sequence

The X server may be terminated by pressing a particular set of keys. By default, that set is left Shift, CTRL, and Reset.

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

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

display specifies the display.

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

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

Enabling the Reset Key Sequence

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

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

display specifies the display.

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

5

X Input Device Extension Functions

The functions described in this chapter allow client programs to access input devices other than the X keyboard and the X pointer.

Note

The functions described in this chapter supersede many of the HP extension functions described in Chapter 6. You should use the functions described in this chapter unless you require functionality that is only supported through the HP extension functions.

Do not mix functions of the two types. If you need to use input device extension functions, select *either* the X standard functions *or* the HP functions.

None of these features are required in order for the X server or X clients to operate correctly if the X keyboard and X pointer are the only input devices.

These input device extension functions are accessible through the library `libXi.a`. Defined constants and structures needed by clients that use these functions are found in the files `XI.h` and `XInput.h`.

X include files are installed in a subdirectory of `/usr/include`. For example, if the HP-UX Developer's Toolkit 1.0 product is installed, the X include files are installed in `/usr/include/X11R5/X11`.

Refer to the sample program at the end of this chapter for more information about using the functions described below.

Listing Available Input Devices

Clients that wish to access input devices through the input extension typically perform the following steps:

- Determine which input devices are available via `XListInputDevices`.
- Open the desired devices via `XOpenDevice`, specifying device ids obtained via `XListInputDevices`.
- Determine the eventclass to be used in selecting each desired input extension event, and the event type that the event will have. This is done using macros provided by the input extension and the `XDevice` structure returned by `XOpenDevice`.
- Select the desired events via the `XSelectExtension` request, passing the eventclasses obtained above.

- Receive the desired events via XNextEvent.

Listing Input Devices

To obtain a list of available input devices, use XListInputDevices.

```
XDeviceInfo *XListInputDevices(display, ndevices_return)
      Display *display;
      int     *ndevices_return;
```

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.

The XListInputDevices function lists the available input devices. This list includes the X pointer, the X keyboard, and any other input devices that are currently accessible through the X server.

Input devices are not opened until requested by some client. After an input device has been listed, it is possible for some non-X process to open that device. In this case, an X request to open a device can fail because the device is no longer available, even though it was available when listed.

For each input device available to the server, the XListInputDevices request returns an XDeviceInfo structure. The inputclassinfo field of that structure contains a pointer to a list of variable-length 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 following constants identify standard device names: XI_MOUSE, XI_TABLET, XI_KEYBOARD, XI_TOUCHSCREEN, XI_TOUCHPAD, XI_BUTTONBOX, XI_BARCODE, XI_TRACKBALL, XI_QUADRATURE, XI_ID_MODULE, XI_ONE_KNOB, and XI_NINE_KNOB.

Additional input devices may be supported in future releases.

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.

Any client may change the use of an input device via the `XChangeKeyboardDevice` or `XChangePointerDevice` requests.

The *inputclassinfo* field contains a pointer to the first input-class specific data. The first two fields are common to all classes. The list of classes supported by each device is a linked list. Refer to the sample program at the end of this chapter for information about traversing that list.

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;
```

The *min_keycode* field specifies the minimum keycode that the device will report. The minimum keycode will not be smaller than 8.

The *max_keycode* field specifies the maximum keycode that the device will report. The maximum keycode will not be larger than 255.

The *num_keys* field specifies the number of keys that the device has.

The `XButtonInfo` structure describes the characteristics of the buttons on the device. It is defined as follows:

```
typedef struct _XButtonInfo {
        XID    class;
        int     length;
        short   num_buttons;
} XButtonInfo
```

The *num_buttons* field specifies the number of buttons that the device has.

The `XValuatorInfo` structure describes the characteristics of the valuators 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;
```

The *num_axes* field contains the number of axes the device supports.

The *mode* field 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.

The *motion_buffer_size* field 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 describes the characteristics of a single valuator on the device. It is defined as follows:

```
typedef struct _XAxisInfo {
        int   resolution;
        int   min_value;
        int   max_value;
} XAxisInfo;
```

The *resolution* field contains a number in counts/meter.

The *min_value* 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_value* 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.

Freeing the List of Input Devices

To free the XDeviceInfo array created by XListInputDevices, use XFreeDeviceList.

```
XFreeDeviceList(list)
      XDeviceInfo *list;
```

list Specifies a pointer to the list of XDeviceInfo structures to be freed.

The XFreeDeviceList function frees the list of available extension input devices.

Enabling and Disabling Input Devices

Opening Extended Input Devices

In order to access input devices through the input extension, clients must request that the server open those devices. To open an extended input device, use XOpenDevice.

```
XDevice *XOpenDevice(display, device_id)
      Display *display;
      XID      device_id;
```

display Specifies the connection to the X server.

device_id Specifies the id of the device to be opened. This id is obtained via the XListInputDevices request.

The XOpenDevice function makes an input device accessible to a client through input extension protocol requests. If successful, it returns a pointer to an XDevice structure.

XOpenDevice can generate a BadDevice error.

The XDevice structure 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 this 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`, `DeviceButtonRelese`, `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 one of these events is received via `XNextEvent`, the type can be used for comparison with the type of the event.

Closing Input Devices

Before terminating, clients that have opened input devices through the input extension should close them. To close an extension input device, use `XCloseDevice`.

```
XCloseDevice(display, device)
      Display *display;
      XDevice *device;
```

display Specifies the connection to the X server.

device Specifies the device to be closed.

The `XCloseDevice` function makes an input device inaccessible to a client through input extension protocol requests. 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 will disable all access by X to the device.

XCloseDevice can generate a BadDevice error.

Selecting Input from Extension Input Devices

Selecting Extension Events

To select input from an extended input device, use XSelectExtensionEvent

```
int XSelectExtensionEvent(display, w, event_list, event_count)
      Display     *display;
      Window       w;
      XEventClass *event_list;
      int          event_count;
```

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.

The XSelectExtensionEvent function requests that the X server 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. To receive these events with automatic passive grabbing, specify both DeviceButtonPress and 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 macros include DeviceKeyPress, DeviceKeyRelease, DeviceButtonPress, DeviceButtonRelese, 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. DeviceKeyPress from other devices will have a different event class since the event class identifies both the event and the device.

XSelectExtensionEvent can generate a BadWindow or BadClass error.

Getting the List of Currently Selected Extension Events

To get the list of currently selected extension events, use XGetSelectedExtensionEvents.

```
int XGetSelectedExtensionEvents(display, w,
this_client_event_count_return, this_client_event_list_return,
all_clients_event_count_return, all_clients_event_list_return)
      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;
```

display Specifies the connection to the X server.

w Specifies the window whose events you are interested in.

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.

The XGetSelectedExtensionEvents function reports the extension events selected by this client and all clients for the specified window.

This function returns pointers to two event class 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.

Sending Extension Events

To send input extension events to a client, use XSendExtensionEvent.

```
Status XSendExtensionEvent(display, device, destination,
propagate, event_count, event_list, event_send)
      Display     *display;
      XDevice     *device;
      Window       destination;
      Bool         propagate;
      int          event_count;
      XEventClass *event_list;
      XEvent      *event_send;
```

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 a 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.

The XSendExtensionEvent 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 class list. For a discussion of the valid event class names, see XOpenDevice(3X11). This function uses the *destination* argument to identify the destination window as follows:

- If *destination* is PointerWindow, the destination window is the window that contains the pointer.

- If *destination* 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_count* is zero, 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_list array.
- If *propagate* is True and no clients have selected from the destination window any of the events in the event_list array, the destination is replaced with the closest ancestor of destination for which some client has selected one of the specified events, 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 *event_list*.

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 BadDevice, BadValue, BadClass, and BadWindow errors.

Getting the dont-propagate-list

Input extension events are propagated to ancestor windows unless some client specifies otherwise.

Grabs of extension input devices may alter the set of windows that receive a particular input extension event.

To determine which events will not be propagated from a given window, use XGetDeviceDontPropagateList.

```
XEventClass *XGetDeviceDontPropagateList(display, window,
count_return)
      Display *display;
      Window   window;
      int      *count_return;
```

display Specifies the connection to the X server.

window Specifies the window whose dont-propagate-list is to be queried.

count_return Returns the number of event classes in the list returned by this request.

The XGetDeviceDontPropagateList function returns a list of input extension events that will not be propagated to ancestors of the event window. An array of event classes is returned that identifies which events will not be propagated.

XGetDeviceDontPropagateList can generate a BadClass or BadWindow error.

You should use XFree to free the data returned by this function.

Changing the dont-propagate-list

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 whose propagation can be suppressed include: DeviceKeyPress, DeviceKeyRelease, DeviceButtonPress, DeviceButtonRelease, DeviceMotionNotify, ProximityIn, and ProximityOut.

To change which events will not be propagated from a given window, use XChangeDeviceDontPropagateList.

```
int XChangeDeviceDontPropagateList(display, window,
count, event_list, mode)
      Display     *display;
      Window       window;
      int          count;
      XEventClass *event_list;
      int          mode;
```

display Specifies the connection to the X server.

window Specifies the window whose dont-propagate-list is to be modified.

count Specifies the number of event classes in the list.

event_list Specifies a pointer to a list of event classes.

mode Specifies the mode. You can pass AddToList, or DeleteFromList.

The XChangeDeviceDontPropagateList function modifies the list of input extension events that should not be propagated to ancestors of the event window. This function 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.

XChangeDeviceDontPropagateList can generate a BadDevice, BadClass, or BadValue error.

Getting Extended Device Motion History

To get the device motion history, use `XGetDeviceMotionEvents`.

```
XDeviceTimeCoord *XGetDeviceMotionEvents(display, device,
start, stop, nevents_return, mode_return, axis_count_return)
      Display *display;
      XDevice *device;
      Time     start, stop;
      int     *nevents_return;
      int     *mode_return;
      int     *axis_count_return;
```

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.

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` function makes this history available.

The `XGetDeviceMotionEvents` function returns all events in the motion history buffer that fall between the specified start and stop times, inclusive. If the start time is later than the stop time or if the start time is in the future, no events are returned. 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). These constants are defined in the file `XI.h`. The *axis_count* returns the number of axes or valuators being reported by the device.

`XGetDeviceMotionEvents` can generate a `BadDevice`, `BadMatch` or `BadWindow` error.

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` structure returned by the `XListInputDevices` function. 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 function.

`XGetDeviceMotionEvents` can generate a `BadDevice` or `BadMatch` error.

Freeing the Device Motion Array

To free the device motion array, use `XFreeDeviceMotionEvents`.

```
XFreeDeviceMotionEvents(events)
     XDeviceTimeCoord *events;
```

events Specifies the pointer to the `XDeviceTimeCoord` array returned by a previous call to `XGetDeviceMotionEvents`.

This function frees the array of motion information.

Grabbing and Ungrabbing Extension Input Devices

Grabbing Extended Input Devices

To grab a specified extension device, use `XGrabDevice`.

```
int XGrabDevice(display, device, grab_window,
owner_events, event_count, event_list, this_device_mode,
other_devices_mode, time)
      Display     *display;
      XDevice     *device;
      Window       grab_window;
      Bool         owner_events;
      int          event_count;
      XEventClass *event_list;
      int          this_device_mode, other_devices_mode;
      Time         time;
```

display Specifies the connection to the X server.

device Specifies the device to be grabbed.

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 the events are 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 by 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`.

The `XGrabDevice` function 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. The *event_list* argument 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*.

The *this_device_mode* parameter controls further processing of events from this device and the *other_device_mode* parameter controls the further processing of events from all other devices.

If *this_device_mode* 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 *this_device_mode* 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.

Ungrabbing Extended Input Devices

To ungrab a specified extension device, use `XUngrabDevice`.

```
int XUngrabDevice(display, device, time)
      Display *display;
      XDevice *device;
      Time     time;
```

display Specifies the connection to the X server.

device Specifies the device to be released.

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

The `XUngrabDevice` function releases the device and any queued events if this client has it actively grabbed from `XGrabDevice`, `XGrabDeviceButton`, 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.

Grabbing Extended Input Device Buttons

To grab extension input device buttons, use XGrabDeviceButton.

```
int XGrabDeviceButton(display, device, button,
modifiers, modifier_device, grab_window,
owner_events, event_count, event_list,
this_device_mode, other_devices_mode)
      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, other_devices_mode;
```

display Specifies the connection to the X server.

device Specifies the device that is to be grabbed.

button Specifies the device button that is to be grabbed or AnyButton.

modifiers Specifies the set of keymasks or AnyModifier. The mask is the bitwise inclusive OR of the valid keymask bits. Valid bits are: ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, Mod5Mask.

modifier_device Specifies the device whose modifiers are to be used. If 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 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.

The XGrabDeviceButton function establishes a passive grab. When the specified button is pressed, 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 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.

If some other client has already issued a XGrabDeviceButton with the same button modifier combination on the same window, a BadAccess error results. If AnyModifier was specified for the *modifiers* argument or AnyButton for the *key* argument, 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.

Ungrabbing Extended Input Device Buttons

To ungrab an extended input device button, use `XUngrabDeviceButton`.

```
int XUngrabDeviceButton(display, device, button,
modifiers, modifier_device, grab_window)
      Display      *display;
      XDevice      *device;
      unsigned int button;
      unsigned int modifiers;
      XDevice      *modifier_device;
      Window        grab_window;
```

display Specifies the connection to the X server.

device Specifies the device that is to be released.

button Specifies the device button that is to be released or `AnyButton`.

modifiers Specifies the set of keymasks or `AnyModifier`. The mask is the bitwise inclusive OR of the valid keymask bits. Valid bits are: `ShiftMask`, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, and `Mod5Mask`.

modifier_device Specifies the device whose modifiers are to be used. If `NULL` is specified, the X keyboard will be used as the modifier device.

grab_window Specifies the grab window.

The `XUngrabDeviceButton` function releases the passive grab for a button modifier 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. `XUngrabDeviceButton` has no effect on an active grab.

`XUngrabDeviceButton` can generate `BadDevice`, `BadMatch`, `BadValue`, and `BadWindow` errors.

Grabbing Extended Input Device Keys

To grab an extension input device key, use `XGrabDeviceKey`.

```
XGrabDeviceKey(display, device, key, modifiers,
modifier_device, grab_window, owner_events, event_count,
event_list, this_device_mode, other_devices_mode)
      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, other_devices_mode;
```

display Specifies the connection to the X server.

device Specifies the device that is to be grabbed.

key Specifies the device key that is to be grabbed or `AnyKey`.

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

modifier_device Specifies the device whose modifiers are to be used. If `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`.

The `XGrabDeviceKey` function 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 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 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 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 keys 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 function. Otherwise, a BadValue error will result.

This request overrides all previous grabs by the same client on the same key 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 *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 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.

Ungrabbing Extended Input Device Keys

To ungrab an extended input device key, use XUngrabDeviceKey.

```
XUngrabDeviceKey(display, device, key, modifiers,
modifier_device, grab_window)
      Display      *display;
      XDevice      *device;
      unsigned int key;
      unsigned int modifiers;
      XDevice      *modifier_device;
      Window        grab_window;
```

display Specifies the connection to the X server.

device Specifies the device that is to be released.

key Specifies the device key that is to be 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 NULL is specified, the X keyboard will be used as the modifier device.

grab_window Specifies the grab window.

The XUngrabDeviceKey function releases the passive 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.

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

Releasing Queued Events

To release queued events, use XAllowDeviceEvents.

```
int XAllowDeviceEvents(display, device, event_mode, time)
      Display *display;
      XDevice *device;
      int      event_mode;
      Time     time;
```

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, SyncAllDevices, or AsyncAllDevices.

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

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 event modes are defined as follows:

`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.

`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. Then the specified device appears to freeze unless the reported event causes the grab to be released. `SyncThisDevice` has no effect if the specified device is not frozen or 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, the grab is released and that event is completely reprocessed. This time, however, the request ignores any passive grabs at or above 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 a 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 of behalf of multiple separate grabs, `AsyncOtherDevices` thaws for all. `AsyncOtherDevices` has no effect if the devices are not frozen by the client.

`SyncAllDevices` If all the devices are frozen by the client, event processing for all the devices continues normally until the next button or key event is reported to the client for a grabbed device. Then the devices appear to freeze unless the reported event causes the grab to be released. If any device is still grabbed, then a subsequent event for it will still cause all the devices to freeze. `SyncAllDevices` has no effect unless all the devices are frozen by the client. If any device is frozen twice by the client on behalf of two separate grabs, `SyncAllDevices` thaws for both. A subsequent freeze for `SyncAllDevices` will only freeze each device once.

`AsyncAllDevices` 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, `AsyncAllDevices` thaws for all. `AsyncAllDevices` 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 `SyncAllDevices` or `AsyncAllDevices`, 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 that device. It is possible for a single device to be frozen because of several grabs. In that case, the freeze must be released on behalf or each grab before events can again be processed.

`XAllowDeviceEvents` can generate a `BadDevice or BadValue` error.

Focusing Extension Input Devices

Getting Extended Input Device Focus

To get the focus for an extended input device, use `XGetDeviceFocus`.

```
XGetDeviceFocus(display, device, focus_return,
revert_to_return, time_return)
        Display *display;
        Display *device;
        Window  *focus_return;
        int     *revert_to_return;
        int     *time_return;
```

display Specifies the connection to the X server.

device Specifies the device whose focus is to be queried.

focus_return Returns the focus window, `PointerRoot`, `FollowKeyboard`, or `None`.

revert_to_return Returns the current focus state `RevertToParent`, `RevertToPointerRoot`, `RevertToFollowKeyboard`, or `RevertToNone`.

time_return Returns the *last-focus-time* for the device.

The `XGetDeviceFocus` function returns the focus window and the current focus state. Not all input extensions 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.

Setting Extended Input Device Focus

To set the focus for an extended input device, use XSetDeviceFocus.

```
int XSetDeviceFocus(display, device, focus, revert_to, time)
      Display *display;
      Display *device;
      Window   focus;
      int      revert_to;
      Time     time;
```

display Specifies the connection to the X server.

device Specifies the device whose focus is to be changed.

focus Specifies the 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.

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

The XSetDeviceFocus function 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 the specified 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 `XI.h`).

`XSetDeviceFocus` can generate `BadDevice`, `BadMatch`, `BadValue`, and `BadWindow` errors.

Controlling Device Encodings

Getting the Key Mapping of Extended Input Devices

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

```
KeySym *XGetDeviceKeyMapping(display, device, first_keycode,
keycode_count, keysyms_per_keycode_return)
      Display *display;
      XDevice *device;
      KeyCode  first_keycode;
      int       keycode_count;
      int      *keysyms_per_keycode_return;
```

display Specifies the connection to the X server.

device Specifies the device whose key mapping is to be queried.

first_keycode	Specifies the first KeyCode to be returned.
keycode_count	Specifies the number of KeyCodes to be returned.
keysyms_per_keycode_return	Returns the number of KeySyms per KeyCode.

For the specified device, the `XGetDeviceKeyMapping` 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 `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 results.

`XGetDeviceKeyMapping` can generate a `BadDevice`, `BadMatch`, or `BadValue` error.

Changing the Key Mapping of Extended Input Devices

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

```
int XChangeDeviceKeyMapping(display, device, first_keycode,
keysyms_per_keycode, keysyms, keycode_count)
      Display *display;
      XDevice *device;
      int      first_keycode;
      int      keysyms_per_keycode;
      KeySym  *keysyms;
      int      keycode_count;
```

display	Specifies the connection to the X server.

device	Specifies the device whose key mapping is to be modified.
first_keycode	Specifies the first KeyCode to be changed.
keysyms_per_keycode	Specifies the number of KeySyms per KeyCode.
keysyms	Specifies the address of an array of KeySyms.
keycode_count	Specifies the number of KeyCodes to be modified.

For the specified device, the `XChangeDeviceKeyMapping` function defines the symbols for the specified number of KeyCodes starting with *first_keycode*. The symbols for KeyCodes outside this range remain unchanged. The number of elements in keysyms must be:

$$num\_codes * keysyms\_per\_keycode$$

The specified *first_keycode* must be greater than or equal to *min_keycode* returned by `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:

$$first\_keycode + num\_codes - 1$$

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

$$(K - first\_keycode) * keysyms\_per\_keycode + N$$

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

Getting the Modifier Mapping of Extended Input Devices

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

```
XModifierKeymap *XGetDeviceModifierMapping(display, device)
      Display *display;
      XDevice *device;
```

display Specifies the connection to the X server.

device Specifies the device whose modifier mapping is to be queried.

The XGetDeviceModifierMapping 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 XFreeModifierMapping. If only zero values appear in the set for any modifier, that modifier is disabled.

XGetDeviceModifierMapping can generate BadDevice and BadMatch errors.

Setting the Modifier Mapping of Extended Input Devices

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

```
int XSetDeviceModifierMapping(display, device, modmap)
      Display          *display;
      XDevice          *device;
      XModifierKeymap  *modmap;
```

display Specifies the connection to the X server.

device Specifies the device whose modifier mapping is to be modified.

modmap Specifies a pointer to the XModifierKeymap structure.

The XSetDeviceModifierMapping function 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 *modmap* 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. Such restrictions are needed, 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 such a 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 are changed.

XSetDeviceModifierMapping can generate BadLength, BadDevice, BadMatch, BadAlloc, and BadValue errors.

The XModifierKeymap structure contains:

```
typedef struct {
        int       max_keypermod;
        KeyCode *modifiermap;
} XModifierKeymap;
```

Querying the Device Button Mapping

To check the device button mapping, use XGetDeviceButtonMapping.

```
int XGetDeviceButtonMapping(display, device, map_return,
nmap)
      Display       *display;
      XDevice       *device;
      unsigned char map_return[];
      int            nmap;
```

display Specifies the connection to the X server.

device Specifies the device whose button mapping is to be queried.

map_return Returns the mapping list.

nmap Specifies the number of items in the mapping list.

The XGetDeviceButtonMapping function 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 and BadMatch errors.

Changing the Device Button Mapping

To change the device button mapping, use `XSetDeviceButtonMapping`.

```
int XSetDeviceButtonMapping(display, device, map, nmap)
      Display      *display;
      XDevice      *device;
      unsigned char map[];
      int           nmap;
```

display Specifies the connection to the X server.

device Specifies the device whose button mapping is to be changed.

map Specifies the mapping list.

nmap Specifies the number of items in the mapping list.

The `XSetDeviceButtonMapping` function 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.

Changing the Core X Devices

Changing the X Keyboard Device

To change the X keyboard device, use `XChangeKeyboardDevice`.

```
Status XChangeKeyboardDevice(display, device)
      Display *display;
      XDevice *device;
```

display Specifies the connection to the X server.

device Specifies the device to be used as the X keyboard.

The `XChangeKeyboardDevice` function causes the server to use the specified device as the X keyboard. The server implementation must support focusing of the new device, or a `BadDevice` error will be returned. Whether or not a given device can be focused can be determined by examining the information returned by the XOpenDevice request. For those devices that can be focused, XOpenDevice will return an XInputClassInfo structure with the *input_class* field equal to the constant FocusClass (defined in the file XI.h).

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.

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.

Changing the X Pointer Device

To change the X pointer device, use XChangePointerDevice.

```
Status XChangePointerDevice(display, device, xaxis, yaxis)
      Display *display;
      XDevice *device;
      int      xaxis;
      int      yaxis;
```

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.

The XChangePointerDevice function causes the server to use the specified device as the X pointer.

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.

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 BadDevice and BadMatch errors.

Feedback Control

These functions are provided to manipulate 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` will return an `XInputClassInfo` structure with the *input_class* field equal to the constant `FeedbackClass` (defined in the file `XI.h`).

Each class of feedback is described by a structure specific to that class. These structures are defined in the file *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 *click* field specifies the key-click volume, with values in the range of 0 (off) to 100 (loud). The *percent* field specifies the bell volume, with a range of 0 (off) to 100 (loud). The *pitch* field specifies the bell pitch in Hz. The range of the value is implementation-dependent. The *duration* field specifies the duration in milliseconds of the bell. The *led_mask* field is a bit mask that describes the current state of up to 32 LEDs. A value of 1 in a bit indicates that the corresponding LED is on. The *global_auto_repeat* field has a value of `AutoRepeatModeOn` or `AutoRepeatModeOff`. The *auto_repeats* field is a bit vector. Each bit set to 1 indicates that the auto-repeat is enabled for the corresponding key.

The `XPtrFeedbackState` structure defines the attributes that are returned for feedbacks equivalent to those on the X pointer.

```
typedef struct {
    XID      class;
    int      length;
    XID      id;
    int      accelNum;
    int      accelDenom;
    int      threshold;
} XPtrFeedbackState;
```

The *accelNum* field returns the numerator for the acceleration multiplier. The *accelDenom* field returns the denominator for the acceleration multiplier. The *threshold* field returns the threshold for the acceleration.

The `XIntegerFeedbackState` structure defines the attributes that are returned for integer feedbacks.

```
typedef struct {
    XID      class;
    int      length;
    XID      id;
    int      resolution;
    int      minVal;
    int      maxVal;
} XIntegerFeedbackState;
```

The *resolution* field specifies the number of digits that the feedback can display. The *minVal* field specifies the minimum value that the feedback can display. The *maxVal* field specifies the maximum value that the feedback can display.

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 *max_symbols* field specifies the maximum number of symbols that can be displayed. The *syms_supported* field is a pointer to the list of supported symbols. The *num_syms_supported* field specifies the length of the list of supported symbols.

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;
```

Bell feedbacks are those that can generate a sound. Some implementations may support a bell as part of a `KbdFeedback` feedback. Class `BellFeedback` is provided for implementations that do not choose to do so, and for devices that support multiple feedbacks that can produce sound. The meaning of the fields is the same as that of the corresponding fields in the `XKbdFeedbackState` structure.

The `XLedFeedbackState` structure defines the attributes that are returned for Led feedbacks.

```
    typedef struct {
        XID      class;
        int      length;
        XID      id;
        int      led_values;
        int      led_mask;
    } XLedFeedbackState;
```

LED feedbacks are those that can generate a light. Up to 32 lights per feedback are supported. Each bit in *led_mask* corresponds to one light, and the corresponding bit in *led_values* indicates whether that light should be on or off. Some implementations may support LEDs as part of a `KbdFeedback` feedback. Class `LedFeedback` is provided for implementations that do not choose to do so, and for devices that support multiple LED feedbacks. The meaning of the *led_values* field is the same as that in the `XKbdFeedbackState` structure.

The `XPtrFeedbackControl` structure defines the attributes that can be controlled for feedbacks equivalent to those on the X pointer.

```
    #define DvAccelnum      (1L<<0)
    #define DvAccelDenom    (1L<<1)
    #define DvThreshold     (1L<<2)
    typedef struct {
        XID      class;
        int      length;
        XID      id;
        int      accelNum;
        int      accelDenom;
        int      threshold;
    } XPtrFeedbackControl;
```

The acceleration, expressed as a fraction, is a multiplier for movement. For example, specifying 3/1 means that the device moves three times as fast as normal. The fraction may be rounded arbitrarily by the X server. Accelerations only takes effect if the device moves more that the 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 *accelNum* and *threshold* fields must be nonzero for the pointer values to be set. Otherwise, the parameter will be unchanged.

The XKbdFeedbackControl structure defines the attributes that can be controlled for feedbacks equivalent to those on the X keyboard.

```
#define         DvKeyClickPercent       (1L<<0)
#define         DvPercent               (1L<<1)
#define         DvPitch                 (1L<<2)
#define         DvDuration              (1L<<3)
#define         DvLed                   (1L<<4)
#define         DvLedMode               (1L<<5)
#define         DvKey                   (1L<<6)
#define         DvAutoRepeatMode        (1L<<7)
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.

```
#define         DvString           (1L<<0)
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.

```
#define         DvInteger        (1L<<0)
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.

```
#define         DvPercent       (1L<<1)
#define         DvPitch         (1L<<2)
#define         DvDuration      (1L<<3)
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.

```
#define         DvLed           (1L<<4)
#define         DvLedMode       (1L<<5)
typedef struct {
    XID     class;
    int     length;
    XID     id;
    int     led_mask;
    int     led_values;
} XLedFeedbackControl;
```

Querying Input Device Feedbacks

To query input device feedbacks, use XGetFeedbackControl.

```
XFeedbackState *XGetFeedbackControl(display, device,
num_feedbacks)
        Display *display;
        XDevice *device;
        int     *num_feedbacks;
```

display Specifies the connection to the X server.

device Specifies the device whose feedbacks are to be queried.

num_feedbacks Specifies an address into which the number of feedbacks supported by the device is to be returned.

The XGetFeedbackControl function 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 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.

Changing Input Device Feedbacks

To change input device feedbacks, use XChangeFeedbackControl.

```
int XChangeFeedbackControl(display, device, mask, control)
        Display          *display;
        XDevice          *device;
        Mask              mask;
        XFeedbackControl *control;
```

display Specifies the connection to the X server.

device Specifies the device whose feedbacks are to be modified.

mask Specifies a mask specific to each type of feedback that describes how the feedback is modified.

control Specifies the address of an XFeedbackControl structure that contains the new values for the feedback.

The XChangeFeedbackControl function 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.

Miscellaneous Functions

Changing the Mode of an Input Device

To change the mode of a device, use XSetDeviceMode.

```
int XSetDeviceMode(display, device, mode)
        Display *display;
        XDevice *device;
        int      mode;
```

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.

The XSetDeviceMode function 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.

Checking the State of an Extension Input Device

To query the state of the keys, buttons, and valuators of an extension input device, use XQueryDeviceState.

```
XDeviceState *XQueryDeviceState(display, device)
        Display *display;
        XDevice *device;
```

display Specifies the connection to the X server.

device Specifies the device whose state is to be queried.

The XQueryDeviceState function queries the state of an input device. The current state of keys and buttons (up or down), and valuators (current value) on the device is reported by this request. Each key or button is represented by a bit in the XKeyState or XButtonState structure that is returned. Valuators on the device report 0 if they are reporting relative information, or the current value if they are reporting absolute information.

XQueryDeviceState can generate a BadDevice error.

The XDeviceState 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;
        unsigned char num_keys;
        char          keys[32];
} XKeyState;
```

The XButtonState structure contains:

```
typedef struct {
        unsigned char class;
        unsigned char length;
        unsigned char num_buttons;
        char          buttons[32];
} XButtonState;
```

Finding the Extension Version

To find the version of the input extension, use XGetExtensionVersion.

```
XExtensionVersion *XGetExtensionVersion(display, name)
        Display *display;
        char    *name;
```

display Specifies the connection to the X server.

name Specifies the extension to be queried.

The XGetExtensionVersion function queries the version of the input extension, and returns an XExtensionVersion structure. You should use XFree to free the XExtensionVersion structure.

This function returns an XExtensionVersion structure.

```
typedef struct {
        int     present;
        short   major_version;
        short   minor_version;
} XExtensionVersion;
```

Ringing a Bell on an Extension Input Device

To ring a bell on a extension input device, use `XDeviceBell`.

```
void XDeviceBell (display, device, feedbackclass, feedbackid,
percent)
          Display *display;
          XDevice *device;
          XID      feedbackclass, feedbackid;
          int      percent;
```

display Specifies the connection to the X server.

device Specifies the desired device.

feedbackclass Specifies the feedbackclass. Valid values are KbdFeedbackClass and BellFeedbackClass.

feedbackid Specifies the id of the feedback that has the bell.

percent Specifies the volume in the range -100 (quiet) to 100 percent (loud).

This function is analogous to the core `XBell` function. It rings the specified bell on the specified input device feedback using the specified volume.

The specified volume is relative to the base volume for the feedback. If the value for the *percent* argument is not in the range -100 to 100 inclusive, a `BadValue` error results.

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

$$base - \frac{base * percent}{100} + percent$$

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

$$base + \frac{base * percent}{100}$$

To change the base volume of the bell, use `ChangeFeedbackControl`.

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

Initializing Valuators on an Input Device

Some devices that report absolute positional data can be initialized to a starting value. Devices that are capable of reporting relative motion or absolute positional data may require that their valuators be initialized to a starting value after the mode of the device is changed to `Absolute`. To initialize the valuators on such a device, use the `SetDeviceValuators` function.

```
Status XSetDeviceValuators (display, device, valuators,
first_valuator, num_valuators)
          Display *display;
          XDevice *device;
          int      *valuators, first_valuator, num_valuators;
```

display Specifies the connection to the X server.

device Specifies the device whose valuators are to be initialized.

valuators Specifies the values to which each valuator is to be set.

first_valuator Specifies the first valuator to be set.

num_valuators Specifies the number of valuators to be set.

This function initializes the specified valuators on the specified extension input device. Valuators are numbered beginning with zero. Only the valuators in the range specified by *first_valuator* and *num_valuators* are set. If the number of valuators supported by the device is less than the expression $first\_valuator + num\_valuators$, a `BadValue` error will result.

If the request succeeds, `Success` is returned. If the specified device is grabbed by some other client, the request will fail and a status of `AlreadyGrabbed` will be returned.

`XSetDeviceValuators` can generate `BadLength`, `BadDevice`, `BadMatch`, and `BadValue` errors.

Getting Input Device Controls

Some input devices support various configuration controls that can be queried or changed by clients. The set of supported controls will vary from one input device to another. Requests to manipulate these controls will fail if either the target X server or the target input device does not support the requested device control.

Each device control has a unique identifier. Information passed with each device control varies in length and is mapped by data structures unique to that device control.

To query a device control, use the `XGetDeviceControl` function.

```
XDeviceControl *XGetDeviceControl (display, device, control)
         Display *display;
         XDevice *device;
         int       control;
```

display Specifies the connection to the X server.

device Specifies the device whose configuration control status is to be returned.

control Identifies the specific device control to be queried.

This request returns the current state of the specified device control. If the target X server does not support that device control, a `BadValue` error is returned. If the specified device does not support that device control, a `BadMatch` error is returned.

If the request is successful, a pointer to a generic `XDeviceState` structure is returned. The information returned varies according to the specified control and is mapped by a structure appropriate for that control. The first two fields are common to all device controls:

```
typedef struct {
      XID     control;
      int     length;
} XDeviceState;
```

The control may be compared to constants defined in the file `XI.h`. Currently defined device controls include `DEVICE_RESOLUTION`.

The information returned for the `DEVICE_RESOLUTION` control is defined in the following `XDeviceResolutionState` structure:

```
typedef struct {
      XID     control;
      int     length;
      int     num_valuators;
      int     *resolutions;
      int     *min_resolutions;
      int     *max_resolutions;
} XDeviceResolutionState;
```

This device control returns a list of valuators and the range of valid resolutions allowed for each. Valuators are numbered beginning with 0. Resolutions for all valuators on the device are returned. For each valuator i on the device, *resolutions[i]* returns the current setting of the resolution, *min_resolutions[i]* returns the minimum valid setting, and *max_resolutions[i]* returns the maximum valid setting.

When this control is specified for a device that has no valuators, `XGetDeviceControl` will fail with a `BadMatch` error.

`XGetDeviceControl` can generate `BadMatch` and `BadValue` errors.

Changing Input Device Controls

Some input devices support various configuration controls that can be changed by clients. Typically, this is done to initialize the device to a known state or configuration. The set of supported controls varies from one input device to another. Requests to manipulate these controls fail if either the target X server or the target input device does not support the requested device control. Setting the device control also fails if the target input device is grabbed by another client, or has been opened by another client and has been set to a conflicting state.

Each device control has a unique identifier. Information passed with each device control varies in length and is mapped by data structures unique to that device control.

To change a device control use `XChangeDeviceControl`.

```
Status XChangeDeviceControl (display, device, control, value)
        Display        *display;
        XDevice        *device;
        XID             control;
        XDeviceControl *value;
```

display Specifies the connection to the X server.

device Specifies the device whose configuration control status is to be modified.

control Identifies the specific device control to be changed.

value Specifies a pointer to an `XDeviceControl` structure that describes which control is to be changed, and how it is to be changed.

This request changes the current state of the specified device control. If the target X server does not support that device control, a `BadValue` error is returned. If the specified device does not support that device control, a `BadMatch` error is returned. If another client has the target device grabbed, a status of `AlreadyGrabbed` will be returned. If another client has the device open and has set it to a conflicting state, a status of `DeviceBusy` is returned.

If the request fails for any reason, the device control will not be changed.

If the request is successful, the device control will be changed and a status of Success will be returned. The information passed varies according to the specified control and is mapped by a structure appropriate for that control. The first two fields are common to all device controls:

```
typedef struct {
      XID     control;
      int     length;
} XDeviceControl;
```

The control may be set using constants defined in the file `XI.h`. Currently defined device controls include `DEVICE_RESOLUTION`.

The information that can be changed by the `DEVICE_RESOLUTION` control is defined in the following `XDeviceResolutionControl` structure:

```
typedef struct {
      XID     control;
      int     length;
      int     first_valuator;
      int     num_valuators;
      int     *resolutions;
} XDeviceResolutionControl;
```

This device control changes the resolution of the specified valuators on the specified extension input device. Valuators are numbered beginning with zero. Only the valuators in the range specified by *first_valuator* and *num_valuators* are set. A value of -1 in the resolutions list indicates that the resolution for this valuator is not to be changed. The *num_valuators* field specifies the number of valuators in the resolutions list.

When this control is specified, `XChangeDeviceControl` fails with a `BadMatch` error if the specified device has no valuators. If a resolution is specified that is not within the range of valid values (as returned by `XGetDeviceControl`) the request will fail with a `BadValue` error. If the number of valuators supported by the device is less than the expression $first\_valuator + num\_valuators$, a `BadValue` error will result.

Sample X Input Device Extension Program

The following sample program, which creates a window and selects input from it, uses the X Input device extension functions to access input devices other than the X pointer and keyboard.

```
/************************************************************************
 *
 * File: xinput.c
 *
 * Sample program to access input devices other than the X pointer and
 * keyboard using the Input Device extension to X.
 * This program creates a window and selects input from it.
 * To terminate this program, press button 1 on any device being accessed
 * through the extension when the X pointer is in the test window.
 *
 * To compile this program, use
 * "cc xinput.c -I/usr/include/X11R5 -L/usr/lib/X11R5 -lXi -lXext -lX11 -o xinput
 */

#include <X11/Xlib.h>
#include <X11/XInput.h>
#include "stdio.h"

main()
    {
    int                 i, j, count, ndevices, devcnt=0, devkeyp, devbutp;
    Display             *display;
    Window              my;
    XEvent              event;
    XDeviceInfoPtr      list, slist;
    XInputClassInfo     *ip;
    XDeviceButtonEvent  *b;
    XEventClass         class[128];
    XDevice             *dev, *opendevs[9];
    XAnyClassPtr        any;
    XKeyInfoPtr         K;

    if ((display = XOpenDisplay ("")) == NULL)
        {
        printf ("No connection to server - Terminating.\n");
        exit(1);
        }
    my = XCreateSimpleWindow (display, RootWindow(display,0), 100, 100,
    100, 100, 1, BlackPixel(display,0), WhitePixel(display,0));
XMapWindow (display, my);
XSync(display,0);

slist=list=(XDeviceInfoPtr) XListInputDevices (display, &ndevices);
for (i=0; i<ndevices; i++, list++)
    {
    any = (XAnyClassPtr) (list->inputclassinfo);
    for (j=0; j<list->num_classes; j++)
        {
        if (any->class == KeyClass)
            {
            K = (XKeyInfoPtr) any;
            printf ("device %s:\n",list->name);
            printf ("num_keys=%d min_keycode=%d max_keycode=%d\n\n",
                K->num_keys,K->min_keycode,K->max_keycode);
            }
        else if (any->class == ButtonClass)
            printf ("device %s num_buttons=%d\n\n",list->name,
                ((XButtonInfoPtr) any)->num_buttons);
        /*
         * Increment 'any' to point to the next item in the linked
         * list.  The length is in bytes, so 'any' must be cast to
         * a character pointer before being incremented.
         */
        any = (XAnyClassPtr) ((char *) any + any->length);
        }
    if (list->use != IsXKeyboard && list->use != IsXPointer)
        {
        dev = XOpenDevice (display, list->id);
        for (ip= dev->classes, j=0; j<dev->num_classes; j++, ip++)
            if (ip->input_class == KeyClass)
                {
                /* This is a macro, the braces are necessary */
                DeviceKeyPress (dev, devkeyp, class[count++]);
                }
            else if (ip->input_class == ButtonClass)
                {
                DeviceButtonPress (dev, devbutp,class[count++]);
                }
        opendevs[devcnt++]=dev;
        }
    }
XSelectExtensionEvent (display, my, class, count);
for (;;)
    {
    XNextEvent (display, &event);
    if (event.type == devkeyp)
        printf ("Device key press event device=%d\n",
            ((XDeviceKeyEvent *) &event)->deviceid);
    else if (event.type == devbutp)
        {
        b = (XDeviceButtonEvent * ) &event;
        printf ("Device button press event device=%d button=%d\n",
            b->deviceid, b->button);
        if (b->button==1)
            break;
        }
    }
for (i=0; i<devcnt; i++)
    XCloseDevice (display, opendevs[i]);
XFreeDeviceList (slist);
}
```

Sample X Input Device Extension Program

6

HP Input Device Extension Functions

Prior to the addition of the X input device extension functions described in Chapter 5, the standard model for the X Window System consisted of a keyboard and a mouse. Although this met the needs of most users, it did not provide a way to easily use multiple input devices at the same time, and it did not accommodate applications in which a mouse was not the most appropriate input device. To provide better integration with products and peripherals available with HP 9000 computers, including HP-HIL input devices, the extensions described in this chapter were added to the X Window System. Later the X Window System standard was extended to include functions similar to the ones described in this chapter.

Note

These functions are maintained for backwards compatibility only. They will be removed at the next major release of HP-UX.

Most of the functionality described in this chapter has been superseded by equivalent functionality in X input device extension functions which are now a part of standard Xlib. Those overlapping functions are described in Chapter 5. Unless your application requires the use of a function described in this chapter, use those X input device extension functions instead.

These input extension functions are accessible through the library `libXhp11.a`. They will work among all networked HP 9000 computers, but may not work with other vendor's systems on the same network.

Refer to the sample program at the end of this chapter for more information about using the functions described below.

The following functions allow client programs to determine what input devices are available, determine information about each device, and access individual devices.

Listing Available Input Devices

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

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

display Specifies the connection to the X server.

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

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

```
typedef struct
    {
    unsigned int      resolution;        /* resolution in counts/meter */
    unsigned short    min_val;           /* min value this axis returns*/
    unsigned short    max_val;           /* max value this axis returns*/
    } XHPaxis_info;
typedef struct
    {
    XID               x_id;              /* device X identifier       */
    char              *name;             /* device name               */
    XHPaxis_info      *axes;             /* pointer to axes array     */
    unsigned short    type;              /* device type               */
    unsigned short    min_keycode;       /* min X keycode from this dev*/
    unsigned short    max_keycode;       /* max X keycode from this dev*/
    unsigned char     hil_id;            /* device HIL identifier     */
    unsigned char     mode;              /* ABSOLUTE or RELATIVE      */
    unsigned char     num_axes;          /* # axes this device has    */
    unsigned char     num_buttons;       /* # buttons on this device  */
    unsigned char     num_keys;          /* # keys on this device     */
    unsigned char     io_byte;           /* I/O descriptor byte for dev*/
    unsigned short    detailed_id;       /* kbd interface + type      */
    unsigned char     pad[6];            /* reserved for future use   */
    } XHPDeviceList;
```

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

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

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

Device Type Names

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

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

Freeing the DeviceList

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

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

list Specifies the `XHPDeviceList` to free.

When `XHPListInputDevices` is called, it allocates memory in which to place the `XHPDeviceList` array. To free this allocated memory, call `XHPFreeDeviceList` with the `XHPDeviceList` *list* pointer as an argument.

Enabling Extended Input Devices

To enable an extended input device, use `XHPSetInputDevice`.

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

display Specifies the connection to the X server.

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

mode Controls the mode to which the device is set. Valid values are `ON|SYSTEM_EVENTS`, `ON|DEVICE_EVENTS`, and `OFF`.

`XHPSetInputDevice` allows a client program to request the server to open a device or to close a device when it is no longer needed. The client may cause input from the device to be merged with input from the X keyboard or X pointer by using the mode `SYSTEM_EVENTS`, or as an individually-selectable device by using the mode `DEVICE_EVENTS`.

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

`XHPSetInputDevice` can generate `BadDevice` and `BadMode` errors. A `BadMode` error is generated if another client has opened the device with a conflicting mode.

Getting the Event Select Mask and Event Type

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

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

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

display Specifies the connection to the X server.

event_constant Specifies the constant corresponding to the extended event you wish to receive.

eventtype Address of a variable into which the server can return the event type for the extended input event.

mask Address of a variable into which the server can return the event mask to use in selecting that event.

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

Event Select Masks

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

`XHPGetExtMask` may return a `BadType` error.

Selecting Input From Extended Input Devices

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

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

display Specifies the connection to the X server.

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

deviceid Specifies the device from which input is desired.

mask Specifies the mask of input events.

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

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

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

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

Grabbing Extended Input Devices

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

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

display Specifies the connection to the X server.

device_id Specifies the ID of the device to grab.

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

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

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

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

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

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

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

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

Ungrabbing Extended Input Devices

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the device to grab.

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

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

`XHPUngrabDevice` can generate a `BadDevice` error.

Grabbing Extended Input Device Buttons

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

button Specifies the code of the button to be grabbed. You can pass either the button or `AnyButton`.

modifiers Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: `ShiftMask`, `LockMask`, `ControlMask`, `Mod1Mask`, `Mod2Mask`, `Mod3Mask`, `Mod4Mask`, and `Mod5Mask`.

You can also pass AnyModifier, which is equivalent to issuing the grab request for all possible modifier combinations (including the combination of no modifiers).

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

owner_events Specifies a boolean value of either True or False.

event_mask Specifies which device events are to be reported to the client. They can be the bitwise inclusive OR of these device mask bits: HPDeviceButtonPressMask, HPDeviceButtonReleaseMask, HPDevicePointerMotionMask, and HPDeviceKeymapStateMask.

pointer_mode Only the constant GrabModeAsync is currently supported.

device_mode Only the constant GrabModeAsync is currently supported.

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

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

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

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

Ungrabbing Extended Input Device Buttons

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

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

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

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

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

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

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

Grabbing Extended Input Device Keys

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

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

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

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

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

pointer_mode Only the constant `GrabModeAsync` is currently supported.

device_mode Only the constant `GrabModeAsync` is currently supported.

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

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

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

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

Ungrabbing Extended Input Device Keys

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

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

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

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

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

XHPUngrabDeviceKey can generate BadDevice and BadWindow errors.

Getting Extended Input Device Focus

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the device to examine.

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

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

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

Setting Extended Input Device Focus

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the extended device.

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

revert_to Specifies which window the input focus reverts to if the window becomes not viewable. You can pass `RevertToParent`, `RevertToPointerRoot`, or `RevertToNone`.

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

The `XHPSetDeviceFocus` function changes the input focus and the *last-focus-change* time. The function has no effect if the specified time is earlier than the current *last-focus-change* time or is later than the current X server time. Otherwise, the *last-focus-change* time is set to the specified time (`CurrentTime` is replaced by the current X server time). This function causes the X server to generate `XHPDeviceFocusIn` and `XHPDeviceFocusOut` events.

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

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

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

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

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

Getting Current Extended Input Event Selection Masks

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

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

display Specifies the connection to the X server.

window Specifies the window ID of the window to examine.

deviceid Specifies the ID of the device to examine.

mask_return Returns the current extended input event mask.

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

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

Getting Extended Device Motion History

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

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

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

display Specifies the connection to the X server.

deviceid Specifies the extended input device.

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

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

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

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

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

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

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

Enabling Auto-Repeat for Extended Input Devices

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

mode Specifies the auto-repeat rate. Valid values are `REPEAT_30`, which causes repeats to take place every $\frac{1}{30}$ of a second, and `REPEAT_60`, which causes repeats to take place every $\frac{1}{60}$ of a second.

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

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

Disabling Auto-Repeat for Extended Input Devices

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

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

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

Sending a Prompt to Extended Input Devices

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

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

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

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

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

Sending an Acknowledge to Extended Input Devices

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

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

display Specifies the connection to the X server.

deviceid Specifies the ID of the desired device.

acknowledge Specifies the acknowledge to be sent. Valid values are: GENERAL_ACKNOWLEDGE, ACKNOWLEDGE_1, ACKNOWLEDGE_2, ACKNOWLEDGE_3, ACKNOWLEDGE_4, ACKNOWLEDGE_5, ACKNOWLEDGE_6, and ACKNOWLEDGE_7.

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

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

XHPAcknowledge can generate BadDevice and BadValue errors.

Getting Control Attributes of Extended Input Devices

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

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

display Specifies the connection to the X server.

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

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

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

XHPGetDeviceControl returns the control attributes of the device in the XHPDeviceState structure, which is defined as follows:

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

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

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

Setting Control Attributes of Extended Input Devices

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

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

display Specifies the connection to the X server.

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

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

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

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

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

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

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

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

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

If both the *led* and *led_mode* members are specified, the state of that LED is changed, if possible. The *led_mode* member can be set to `LedModeOn` or `LedModeOff`. If only *led_mode* is specified, the state of all LEDs are changed, if possible. At most, 32 LEDs (counting from one) are supported. No standard interpretation of LEDs is defined. If an *led* is specified without an *led_mode*, a `BadMatch` error is generated.

If both the *auto_repeat_mode* and key members are specified, the key and *auto_repeat_mode* members are specified, the *auto_repeat_mode* of that key is changed according to `AutoRepeatModeOn`, `AutoRepeatModeOff`, or `AutoRepeatModeDefault`, if possible. If only *auto_repeat_mode* is specified, the global *auto_repeat* mode for the entire device is changed and does not affect the per_key settings. If a key is specified without an *auto_repeat_mode*, a `BadMatch` error is generated.

Getting the Key Mapping of Extended Input Devices

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

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

display Specifies the connection to the X server.

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

first_keycode_wanted Specifies the first keycode to be returned.

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

keysyms_per_keycode_return Specifies the number of keysyms per keycode.

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

Starting with *first_keycode_wanted*, `XHPGetDeviceKeyMapping` returns the symbols for the specified number of KeyCodes. The specified *first_keycode* counted must be greater than or equal to *min_keycode* as reported by the XHPListInputDevices request. Also,

$max\_keycode$ must be greater than $first\_keycode + keycode\_count - 1$. If either of these conditions is not met, the function returns a `BadValue` error. The number of elements in the KeySyms list is: $keycode\_count * keysyms\_per\_code + N$.

KeySym number N (counting from zero) for
KeyCode K has the following index in keysyms:
$(K - first\_keycode\_wanted) * keysyms\_per\_keycode\_return + N$.

The specified *keysyms_per_keycode_return* can be chosen arbitrarily by the client to be large enough to hold all desired symbols. Using the special KeySym value of `NoSymbol` fills in unused elements for individual KeyCodes.

Use XFree to free the returned KeySym list when it is no longer needed.

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

Changing the Key Mapping of Extended Input Devices

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

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

display Specifies the connection to the X server.

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

first_keycode Specifies the first keycode that is to be changed.

keysyms_per_keycode Specifies the number of keysyms per keycode.

keysyms Specifies a pointer to an array of keysyms that are to be used.

num_codes Specifies the number of keycodes that are to be changed. `XHPDeviceState` structure in which the device values will be returned.

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

Starting with *first_keycode*, `XHPChangeDeviceKeyMapping` defines the symbols for the specified number of keycodes. The symbols for keycodes outside this range remain unchanged. The number of elements must be: $num\_codes * keysyms\_per\_keycode$. (Otherwise, a `BadLength` error is generated.)

The specified *first_keycode* must be greater than or equal to min_keycode as reported by the XHPListInputDevices request. Also, *max_keycode* must be greater than $first\_keycode + (num\_codes/keysyms\_per\_keycode) - 1$. If either of these conditions is not met, the function returns a `BadValue` error.

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

$$(K - first\_keycode) * keysyms\_per\_keycode + N.$$

The specified *keysyms_per_keycode* can 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. `NoSymbol` may appear in nontrailing positions of the effective list for a KeyCode. `XHPChangeDeviceKeyMapping` generates a `MappingNotify` event.

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

Setting the Modifier Mapping of Extended Input Devices

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

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

display Specifies the connection to the X server.

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

modmap Specifies a pointer to an XModifierKeymap structure.

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

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

There are eight modifiers, and the modifiermap member of the XModifierKeymap structure contains eight sets of max_keypermod KeyCodes, one for each modifier in the order Shift, Lock, Control, Mod1, Mod2, Mod3, Mod4, and Mod 5. Only nonzero KeyCodes have meaning in each set (zero KeyCodes are ignored). If a nonzero KeyCode is given outside the range specified by *min_keycode* and *max_keycode* as returned by XHPListInputDevices, or a KeyCode appears more than once in the entire map, a BadValue error is generated.

An X server can impose restrictions on how modifiers can be changed (for example, if certain keys do not generate up transitions in hardware or if multiple modifier keys are not supported). If such a restriction is violated, the status reply is MappingFailed, and none of the modifiers are changed. If the new KeyCodes specified for a modifier differ from those currently defined and any (current or new) keys for that modifier are in the logically down state, the status reply is MappingBusy, and no modifier is changed. XHPSetDeviceModifierMapping generates a HPDeviceMappingNotify event when it returns MappingSuccess.

XHPSetDeviceModifierMapping can generate BadDevice, BadLength, and BadValue errors.

Getting the Modifier Mapping of Extended Input Devices

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

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

display Specifies the connection to the X server.

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

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

XHPGetDeviceModifierMapping returns a newly created XModifierKeymap structure that contains the keys being used as modifiers for the specified device. The structure should be freed after use by calling XFreeModifiermap. If only zero values appear in the set for any modifier, that modifier is disabled.

XHPGetDeviceModifierMapping can generate a BadDevice error.

Getting the Server Mode

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

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

display Specifies the connection to the X server.

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

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

Server Modes

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

These constants can be obtained by including the file `<X11/XHPproto.h>`.

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

Sample Use of HP Input Extensions

The following sample program, which creates a window and selects input from it, uses the HP Input device extension functions to access input devices other than the X pointer and keyboard.

Note

The functions used in this example are supported for compatibility with earlier versions of HP Xlib. Refer to the “Sample X Input Device Extension Program” in Chapter 5 for a sample program that uses the newer X standard input extension functions.

```
/************************************************************************
 *
 * File: hpinput.c
 *
 * Sample program to access input devices other than the X pointer and
 * keyboard using the HP extension to X.
 * This program creates a window and selects input from it.
 * To terminate this program, press button 1 on any device being accessed
 * through the extension.
 *
 * To compile this program, use
 * "cc hpinput.c I/usr/include/X11R5 -L/usr/lib/X11R5 -lXhp11 -lX11 -o hpinput
 */

#include <X11/Xlib.h>
#include <X11/XHPlib.h>
#include "stdio.h"

main()
    {
    Display             *display;
    XHPDeviceList       *list, *slist;
    int                 i, ndevices, devkeyp, devbutp;
    Window              my;
    XEvent              event;
    Mask                mask, tmask;
    XHPDeviceButtonEvent *b;

    if ((display = XOpenDisplay ("")) == NULL)
        {
        printf ("No connection to server - Terminating.\n");
        exit(1);
        }
    my = XCreateSimpleWindow (display, RootWindow(display,0), 100, 100,
        100, 100, 1, BlackPixel(display,0), WhitePixel(display,0));
    XMapWindow (display, my);
    XSync(display,0);
    XHPGetExtEventMask (display, HPDeviceKeyPressreq, &devkeyp, &mask);
    XHPGetExtEventMask (display,HPDeviceButtonPressreq,&devbutp,&tmask);
    mask |= tmask;
    slist = list = XHPListInputDevices (display, &ndevices);
    for (i=0; i<ndevices; i++, list++)
        {
        printf ("\nDevice %s has %d keys and %d buttons\n",
            list->name,list->num_keys,list->num_buttons);
        if (list->x_id != XPOINTER && list->x_id != XKEYBOARD)
            {
            XHPSetInputDevice (display, list->x_id, (ON | DEVICE_EVENTS));
            XHPSelectExtensionEvent (display, my, list->x_id, mask);
            }
        }
    for (;;)
    {
    XNextEvent (display,&event);
    if (event.type == devkeyp)
        printf ("Device key press event device=%d\n",
            ((XHPDeviceKeyEvent * ) &event)->deviceid);
    else if (event.type == devbutp)
        {
        b = (XHPDeviceButtonEvent * ) &event;
        printf ("Device button press event device=%d\n", b->deviceid);
        if (b->ev.button==1)
            {
            for (i=0,list=slist; i<ndevices; i++,list++)
                if (list->x_id != XPOINTER && list->x_id != XKEYBOARD)
                    XHPSetInputDevice (display, list->x_id, OFF);
            break;
            }
        }
    }
XHPFreeDeviceList (slist);
}
```

Sample HP Input Device Extension Program

7

Internationalization Support

An internationalized application is adaptable to the requirements of different native languages, local customs, and character string encodings. The process of adapting the operation to a particular native language, local custom, or string encoding is called localization. A goal of internationalization is to permit localization without program source modifications or recompilation.

Release 5 of X11 Xlib provides support for standard routines for the input and output of internationalized text. In all cases this standard functionality should be used instead of the HP proprietary mechanisms explained in this chapter. The functions described in this chapter are provided for backwards compatibility and will be deleted in a future release.

Internationalization in Xlib is based on the concept of a **locale**. A locale defines the "localized" behavior of a program at run-time. Locales affect Xlib in the following ways:

- Encoding and processing of input method text.
- Encoding of resource files and values.
- Encoding and imaging of text strings.
- Encoding and decoding for inter-client text communication.

Xlib provides support for localized text imaging and text input. Sets of functions are provided for multibyte ("char&*") text as well as wide character ("wchar_t") text in the form supported by the host C language environment.

To get this functionality, it is necessary for the client to call either `setlocale()` or `XtSetLanguageProc()` to initialize the clients locale data base. If the client wishes to display localized title strings with Motif's window manager (mwm), then `XtSetLanguageProc()` should be used instead of `setlocale()`.

Controlling Keyboard Input Using HP's X Window System

The X Window System uses the concept of keysyms to control the mapping of keys into characters. The set of keysyms for a particular keyboard is organized into a table called the keymap. To get information about keyboard mapping or to set the keyboard mapping use the `xmodmap` command.

Mapping keyboard for both Extend-char and Meta

A common problem reported by people using HP's X Window System is the conflict between the use of the "extend-char" key to access the extended characters of "Roman8" or "Latin1" with HP's keyboards and the use of the "extend-char" key as a Meta key.

The default mapping is that both keys serve both purposes. However, since HP-UX 9.* it is possible to configure the keyboard so that one key is used as the "extend-char" key and the other as the Meta key.

The "xmodmap" command can be used to inquire and set the mapping for keys on the keyboard. Run the following command.

```
xmodmap -pm
```

For a US or West European keyboard in the default state, this prints:

```
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift       Shift_R (0xc),  Shift_L (0xd)
lock        Caps_Lock (0x37)
control     Control_L (0xe)
mod1        Meta_R (0xa),  Meta_L (0xb),  Mode_switch (0x36)
mod2
mod3
mod4
mod5
```

The `mod1` modifier has entries for both Meta keysyms and for `Mode_switch` as well; and this creates a problem. The solution is to use `mod2` for Mode_switch and change the `Meta_L` key into the `Mode_switch` key. To do this, use "xmodmap" and execute the following command:

```
xmodmap mods
```

where `mods` contains the following four lines:

```
remove Mod1 = Meta_L Mode_switch
keysym Mode_switch = NoSymbol
keysym Meta_L = Mode_switch
add Mod2 = Mode_switch
```

The entries in the file need to be in this order. Again, type:

```
xmodmap -pm
```

The results should be:

```
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift       Shift_R (0xc),  Shift_L (0xd)
lock        Caps_Lock (0x37)
control     Control_L (0xe)
mod1        Meta_R (0xa)
mod2        Mode_switch (0xb)
mod3
mod4
mod5
```

The keyboard then uses the left extend-char key for extended characters and the right extend-char key for Meta. The client must be linked against R4 or R5 Xlib for this to work.

Dead Key Compose processing

HP's X Window System has supported dead key compose processing for HP workstations for some time. This capability is now supported for non-HP servers (workstations and X-terminals) connected to HP systems.

In this form of compose processing a mute (or dead) key is struck followed by a second key. The initial key is a diacritic and the second key is the ASCII character to which the diacritic is to be applied. The diacritic character must be a special muting keysym to initiate the dead-key compose processing. The list of keysym names and the diacritic character to which they apply follows.

keysym name	diacritic character
hpmute_acute	
hpmute_grave	
hpmute_asciicircum	^
hpmute_diaeresis	
hpmute_asciitilde	~
acute	
diaeresis	

To find out which muting diacritics are supported by a keyboard type:

```
xmodmap -pk
```

The entries in the third and fourth column of the keymap are the extend and shift-extend characters.

To set the mute keysyms as they are for HP series 700 terminals execute:

```
xmodmap mutes
```

where `mutes` is a file containing the following five lines.

```
keysym r = r R hpmute_acute
keysym t = t T hpmute_grave
keysym y = y Y hpmute_asciicircum
keysym u = u U hpmute_diaeresis
keysym i = i I hpmute_asciitilde
```

This is the default condition for HP's ITF keyboards.

Multi-key Compose processing

Since HP-UX 9.*, HP's X Window System supports a form of compose processing that can be done using only ASCII characters. To use this form of compose processing, set a keysym to the Multi_key keysym. For example the "Enter/Print" key on an ITF keyboard could be used as the Multi_key. To do this, execute the following command:

```
xmodmap -e keysym Execute = Multi_key
```

Then, compose processing can be done by typing the Multi_key ("Print") followed by two other keys. One key should be the ASCII key that corresponds to one of the diacritic symbols and the other key should be the ASCII character to which the diacritic should be applied. The two keys can be typed in any order. For example, typing "Print ' e" generates a null character. The table of ASCII characters and the diacritics they are used for follows

ASCII character	diacritic character
'	
`	
^	^
:	
"	

~ ~

Input Method Support

The phrase **input method** is used in this chapter to describe whatever mechanism is used to convert keystrokes into characters. Input methods are described in the *Xlib Reference Manual*.

The input methods for most languages supported by HP are simple input methods and require no additional support beyond that provided by Xlib. However, the Asian languages which are supported (Japanese, Korean, Simplified Chinese and Traditional Chinese) require an input server to fully support these languages. Input servers for languages that require them are bundled with that language's version of HP-UX.

X11 R5 provides two sample implementations of input method support. These vary in the protocol used to communicate with an input server. These are the Xsi implementation and the Ximp implementation. HP 's R5 Xlib provides support for both of these protocols as well as an HP-proprietary protocol. The details of these protocols are only of interest to input server developers. Descriptions of the first two protocols are available from the X Consortium. For NLIO, descriptions and a library for input server developers is available as part of the OpenNLIO product.

Use of Asian Input Method Servers

Users who wish to select from among multiple input servers available on a system may set the input method modifier. This can be set using the XmNinputMethod resource for Motif 1.2 applications, or the XMODIFIERS environment variable for non-Motif 1.2 applications. If the value is _HPNLIO, then use of an NLIO-style input method is indicated. This is also the default if no value is specified. If the value of the input method modifier is _XIM_INPUTMETHOD, then an attempt is made to connect to a server using the Xsi input method protocol. If the modifier begins with _XIMP, then an attempt is made to connect to the input method using the Ximp protocol string for connecting to the input method. The following Ximp string is used to determine language and codeset. If the character "#" is encountered in the modifier string, this is changed to "@" to allow connection with input servers on remote machines. To find out what input methods are available on your system, talk to your system administrator.

In general, the capabilities provided by XOpenIM, XCreateIC, XmbLookupString, etc. should replace the functionality provided by XHPConvertLookup and its associated routines. Application developers are encouraged to use these new routines. Support for XHPConvertLookup etc. is provided to assure backward compatibility for existing applications and will be removed from the library in the next major release of HP-UX.

In addition to the IC values that are part of the X Windows System standard, HP supports the following additional IC value:

XNHPNlioctl
: This value is a write-only IC value which performs any of the operations supported as part of XHPNlioctl. The argument passed to XSetICValues is of type XhpNlioCmd. cmd is the element used as the cmd argument for XHPNlioctl, arg is the arg element, and ret is the return value. Setting this IC value is equivalent to calling XHPNlioctl. It should be used when the programmer is using IC's to control input.

Internationalized Output

X11 R5 provides support for internationalized output through the use of font sets, which are accessed through `XCreateFontSet` and its associated routines. That X standard capability should be used instead of the associate font mechanism explained in this chapter.

The associate font mechanism explained here was provided by HP to support internationalized text output before the X standard supported this functionality. The HP associate font mechanism is provided to maintain compatibility with software that still uses the HP associate font mechanism for internationalized output. However, this mechanism will be removed from the library at the next major release of HP-UX.

Associate Font Support

Xlib provides transparent text handling capability, including mixed 8-bit and 16-bit characters, for the following six Xlib functions:

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

In order to allow these functions to support mixed 8-bit and 16-bit characters, the following functions will concurrently load and unload separate 8-bit (font) and 16-bit (associate font) files.

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

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

1. There exists a language designation in the specified font.

   The XLoadFont and XLoadQueryFont functions look for the language designation in the following order:

   - First examine the value of the font property LANGUAGE. This is an 8-bit STRING type property.
   - Next examine the value of the environment variable LANG. Currently, japanese, japanese.euc, korean, chinese-s, and chinese-t are supported as valid LANGUAGE property or LANG environment variable designations.

2. There exists the associate font designation in the specified font.

   XLoadFont and XLoadQueryFont look for the associate font in the following order:

   - First examine the value of the font property ASSOCIATE_FONT. This is an 8-bit STRING type property.
   - Next examine the value of the environment variable XASSOCFONT.

In summary, XLoadFont and XLoadQueryFont look for the font properties LANGUAGE and ASSOCIATE_FONT in the specified font first. If either or both are undefined, then the environment variables LANG and XASSOCFONT are examined instead.

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

Getting the Associate Font

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

```
XFontStruct *XHPGet16bitMixedFontStruct(font)
            Font font;
```

font Specifies the font ID.

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

The XFontStruct structure returned by this function may not be freed.

Checking for 16-bit Characters

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

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

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

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

byte2 Specifies the second byte of a 16-bit character.

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

This function should not be called for EUC data in HP-UX 10.0 or later releases, since EUC characters can then consist of 24-bit or 32-bit values. Results of this routine on such data is undefined.

Conversions Between X11 Keysyms and HP Roman 8 Codes

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

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

keysym Specifies an X11 KeySym.

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

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

Some Keysyms are unique to Hewlett-Packard equipment because Roman 8 contains characters that were not encoded in the Keysyms distributed by MIT.

To convert an HP Roman 8 character into an X11 KeySym, use `XHPRoman8ToKeysym`.

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

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

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