X Input Device Extension Library

Mark Patrick, Ardent Computer
George Sachs
Hewlett-Packard

X Input Device Extension Library

by Mark Patrick
George Sachs
Hewlett-Packard

X Version 11, Release 7.7

Version 1.0

Copyright © 1989, 1990, 1991 Hewlett-Packard Company, Ardent Computer

Permission to use, copy, modify, and distribute this documentation for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. Ardent and Hewlett-Packard make no representations about the suitability for any purpose of the information in this document. It is provided "as is" without express or implied warranty.

Copyright © 1989, 1990, 1991, 1992 X Consortium

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

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

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

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

X Window System is a trademark of The Open Group.

Table of Contents

Chapter 1. Input Extension

Overview

This document describes an extension to the X11 server. The purpose of this extension is to support the use of additional input devices beyond the pointer and keyboard devices defined by the core X protocol. This first section gives an overview of the input extension. The following sections correspond to chapters 9, 10, and 11, ``Window and Session Manager Functions'', ``Events'', and ``Event Handling Functions'' of the ``Xlib - C Language Interface'' manual and describe how to use the input device extension.

Design Approach

The design approach of the extension is to define functions and events analogous to the core functions and events. This allows extension input devices and events to be individually distinguishable from each other and from the core input devices and events. These functions and events make use of a device identifier and support the reporting of *n*-dimensional motion data as well as other data that is not currently reportable via the core input events.

Core Input Devices

The X server core protocol supports two input devices: a pointer and a keyboard. The pointer device has two major functions. First, it may be used to generate motion information that client programs can detect. Second, it may also be used to indicate the current location and focus of the X keyboard. To accomplish this, the server echoes a cursor at the current position of the X pointer. Unless the X keyboard has been explicitly focused, this cursor also shows the current location and focus of the X keyboard.

The X keyboard is used to generate input that client programs can detect.

The X keyboard and X pointer are referred to in this document as the *core devices*, and the input events they generate (`KeyPress` , `KeyRelease` , `ButtonPress` , `ButtonRelease` , and `MotionNotify`) are known as the *core input events*. All other input devices are referred to as *extension input devices*, and the input events they generate are referred to as *extension input events*. This input extension does not change the behavior or functionality of the core input devices, core events, or core protocol requests, with the exception of the core grab requests. These requests may affect the synchronization of events from extension devices. See the explanation in the section titled ``Event Synchronization and Core Grabs.''

Selection of the physical devices to be initially used by the server as the core devices is left implementation dependent. Functions are defined that allow client programs to change which physical devices are used as the core devices.

Extension Input Devices

The input extension controls access to input devices other than the X keyboard and X pointer. It allows client programs to select input from these devices independently from each other and independently from the core devices. Input events from these devices are of extension types (`DeviceKeyPress`, `DeviceKeyRelease`, `DeviceButtonPress`, `DeviceButtonRelease`, `DeviceMotionNotify`, and so on) and contain a device identifier so that events of the same type coming from different input devices can be distinguished.

Extension input events are not limited in size by the size of the server 32-byte wire events. Extension input events may be constructed by the server sending as many wire-sized events as necessary to return the information required for that event. The library event reformatting routines are responsible for combining these into one or more client XEvents.

Any input device that generates key, button, or motion data may be used as an extension input device. Extension input devices may have zero or more keys, zero or more buttons, and may report zero or more axes of motion. Motion may be reported as relative movements from a previous position or as an absolute position. All valuators reporting motion information for a given extension input device must report the same kind of motion information (absolute or relative).

This extension is designed to accommodate new types of input devices that may be added in the future. The protocol requests that refer to specific characteristics of input devices organize that information by *input device classes*. Server implementors may add new classes of input devices without changing the protocol requests.

All extension input devices are treated like the core X keyboard in determining their location and focus. The server does not track the location of these devices on an individual basis and, therefore, does not echo a cursor to indicate their current location. Instead, their location is determined by the location of the core X pointer. Like the core X keyboard, some may be explicitly focused. If they are not explicitly focused, their focus is determined by the location of the core X pointer.

Input Device Classes

Some of the input extension requests divide input devices into classes based on their functionality. This is intended to allow new classes of input devices to be defined at a later time without changing the semantics of these functions. The following input device classes are currently defined:

KEY	The device reports key events.
BUTTON	The device reports button events.
VALUATOR	The device reports valuator data in motion events.
PROXIMITY	The device reports proximity events.
FOCUS	The device can be focused.
FEEDBACK	The device supports feedbacks.

Additional classes may be added in the future. Functions that support multiple input classes, such as the `XListInputDevices` function that lists all available input devices, organize the data they return by input class. Client programs that use these functions should not access data unless it matches a class defined at the time those clients were compiled. In this way, new classes can be added without forcing existing clients that use these functions to be recompiled.

Using Extension Input Devices

A client that wishes to access an input device does so through the library functions defined in the following sections. A typical sequence of requests that a client would make is as follows:

- `XListInputDevices` - lists all of the available input devices. From the information returned by this request, determine whether the desired input device is attached to the server. For a description of the `XListInputDevices` request, see the section entitled ``Listing Available Devices.''

- `XOpenDevice` - requests that the server open the device for access by this client. This request returns an `XDevice` structure that is used by most other input extension requests to identify the specified device. For a description of the `XOpenDevice` request, see the section entitled ``Enabling and Disabling Extension Devices.''

- Determine the event types and event classes needed to select the desired input extension events, and identify them when they are received. This is done via macros whose name corresponds to the desired event, for example, `DeviceKeyPress` . For a description of these macros, see the section entitled ``Selecting Extension Device Events.''

- `XSelectExtensionEvent` - selects the desired events from the server. For a description of the `XSelextExtensionEvent` request, see the section entitled ``Selecting Extension Device Events.''

- `XNextEvent` - receives the next available event. This is the core `XNextEvent` function provided by the standard X libarary.

Other requests are defined to grab and focus extension devices, to change their key, button, or modifier mappings, to control the propagation of input extension events, to get motion history from an extension device, and to send input extension events to another client. These functions are described in the following sections.

Library Extension Requests

Extension input devices are accessed by client programs through the use of new protocol requests. The following requests are provided as extensions to Xlib. Constants and structures referenced by these functions may be found in the files `<X11/extensions/XI.h>` and `<X11/extensions/XInput.h>`, which are attached to this document as Appendix A.

The library will return `NoSuchExtension` if an extension request is made to a server that does not support the input extension.

Input extension requests cannot be used to access the X keyboard and X pointer devices.

Window Manager Functions

This section discusses the following X Input Extension Window Manager topics:

- Changing the core devices
- Event synchronization and core grabs
- Extension active grabs
- Passively grabbing a key
- Passively grabbing a button
- Thawing a device
- Controlling device focus
- Controlling device feedback
- Ringing a bell on an input device
- Controlling device encoding
- Controlling button mapping
- Obtaining the state of a device

Changing the Core Devices

These functions are provided to change which physical device is used as the X pointer or X keyboard.

Note

Using these functions may change the characteristics of the core devices. The new pointer device may have a different number of buttons from the old one, or the new keyboard device may have a different number of keys or report a different range of keycodes. Client programs may be running that depend on those characteristics. For example, a client program could allocate an array based on the number of buttons on the pointer device and then use the button numbers received in button events as indices into that array. Changing the core devices could cause such client programs to behave improperly or to terminate abnormally if they ignore the `ChangeDeviceNotify` event generated by these requests.

These functions change the X keyboard or X pointer device and generate an `XChangeDeviceNotify` event and a `MappingNotify` event. The specified device becomes the new X keyboard or X pointer device. The location of the core device does not change as a result of this request.

These requests fail and return `AlreadyGrabbed` if either the specified device or the core device it would replace are grabbed by some other client. They fail and return `GrabFrozen` if either device is frozen by the active grab of another client.

These requests fail with a `BadDevice` error if the specified device is invalid, has not previously been opened via `XOpenDevice` , or is not supported as a core device by the server implementation.

Once the device has successfully replaced one of the core devices, it is treated as a core device until it is in turn replaced by another `ChangeDevice` request or until the server terminates. The termination of the client that changed the device will not cause it to change back. Attempts to use the `XCloseDevice` request to close the new core device will fail with a `BadDevice` error.

To change which physical device is used as the X keyboard, use the `XChangeKeyboardDevice` function. The specified device must support input class `Keys` (as reported in the `ListInputDevices` request) or the request will fail with a `BadMatch` error.

`int` **`XChangeKeyboardDevice`**`(` *`*display`*`,` *`*device`*`);`

display	Specifies the connection to the X server.
device	Specifies the desired device.

If no error occurs, `XChangeKeyboardDevice` returns `Success` . A `ChangeDeviceNotify` event with the request field set to `NewKeyboard` is sent to all clients selecting that event. A `MappingNotify` event with the request field set to `MappingKeyboard` is sent to all clients. The requested device becomes the X keyboard, and the old keyboard becomes available as an extension input device. The focus state of the new keyboard is the same as the focus state of the old X keyboard.

`XChangeKeyboardDevice` can generate `AlreadyGrabbed` , `BadDevice` , `BadMatch` , and `GrabFrozen` errors.

To change which physical device is used as the X pointer, use the `XChangePointerDevice` function. The specified device must support input class `Valuators` (as reported in the `XListInputDevices` request) and report at least two axes of motion, or the request will fail with a `BadMatch` error. If the specified device reports more than two axes, the two specified in the xaxis and yaxis arguments will be used. Data from other valuators on the device will be ignored.

If the specified device reports absolute positional information, and the server implementation does not allow such a device to be used as the X pointer, the request will fail with a `BadDevice` error.

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

display	Specifies the connection to the X server.
device	Specifies the desired device.
xaxis	Specifies the zero-based index of the axis to be used as the x-axis of the pointer device.
yaxis	Specifies the zero-based index of the axis to be used as the y-axis of the pointer device.

If no error occurs, `XChangePointerDevice` returns `Success` . A `ChangeDeviceNotify` event with the request field set to `NewPointer` is sent to all clients selecting that event. A `MappingNotify` event with the request field set to `MappingPointer` is sent to all clients. The requested device becomes the X pointer, and the old pointer becomes available as an extension input device.

`XChangePointerDevice` can generate `AlreadyGrabbed` , `BadDevice` , `BadMatch` , and `GrabFrozen` errors.

Event Synchronization and Core Grabs

Implementation of the input extension requires an extension of the meaning of event synchronization for the core grab requests. This is necessary in order to allow window managers to freeze all input devices with a single request.

The core grab requests require a pointer_mode and keyboard_mode argument. The meaning of these modes is changed by the input extension. For the `XGrabPointer` and `XGrabButton` requests, pointer_mode controls synchronization of the pointer device, and keyboard_mode controls the synchronization of all other input devices. For the `XGrabKeyboard` and `XGrabKey` requests, pointer_mode controls the synchronization of all input devices, except the X keyboard, while keyboard_mode controls the synchronization of the keyboard. When using one of the core grab requests, the synchronization of extension devices is controlled by the mode specified for the device not being grabbed.

Extension Active Grabs

Active grabs of extension devices are supported via the `XGrabDevice` function in the same way that core devices are grabbed using the core `XGrabKeyboard` function, except that an extension input device is passed as a function parameter. The `XUngrabDevice` function allows a previous active grab for an extension device to be released.

Passive grabs of buttons and keys on extension devices are supported via the `XGrabDeviceButton` and `XGrabDeviceKey` functions. These passive grabs are released via the `XUngrabDeviceKey` and `XUngrabDeviceButton` functions.

To grab an extension device, use the `XGrabDevice` function. The device must have previously been opened using the `XOpenDevice` function.

```
int XGrabDevice( *display, *device, grab_window, owner_events, event_count, *event_list, this_device_mode, other_device_mode, time);
```

"*display*"	Specifies the connection to the X server.
device	Specifies the desired device.
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_count	Specifies the number of elements in the event_list array.
event_list	Specifies a pointer to a list of event classes that indicate which events the client wishes to receive. These event classes must have been obtained using the device being grabbed.
this_device_mode	Controls further processing of events from this device. You can pass one of these constants: `GrabModeSync` or `GrabModeAsync` .

other_device_mode	Controls further processing of events from all other devices. You can pass one of these constants: `GrabModeSync` or `GrabModeAsync` .
time	Specifies the time. This may be either a timestamp expressed in milliseconds or `CurrentTime` .

`XGrabDevice` actively grabs an extension input device and generates `DeviceFocusIn` and `DeviceFocusOut` events. Further input events from this device are reported only to the grabbing client. This function overrides any previous active grab by this client for this device.

The event_list parameter 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` , input events from this device are reported with respect to grab_window and are reported only if specified in event_list. If owner_events is `True` , then if a generated 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 event_list.

The this_device_mode argument controls the further processing of events from this device, and the other_device_mode argument controls the further processing of input events from all other devices.

- If the this_device_mode argument is `GrabModeAsync` , device event processing continues normally; if the device is currently frozen by this client, then processing of device events is resumed. If the this_device_mode argument is `GrabModeSync` , the state of the grabbed device (as seen by client applications) appears to freeze, and no further device events are generated by the server until the grabbing client issues a releasing `XAllowDeviceEvents` call or until the device grab is released. Actual device input events are not lost while the device is frozen; they are simply queued for later processing.

- If the other_device_mode is `GrabModeAsync` , event processing from other input devices is unaffected by activation of the grab. If other_device_mode is `GrabModeSync` , the state of all devices except the grabbed device (as seen by client applications) appears to freeze, and no further events are generated by the server until the grabbing client issues a releasing `XAllowEvents` or `XAllowDeviceEvents` call or until the device grab is released. Actual events are not lost while the other devices are frozen; they are simply queued for later processing.

`XGrabDevice` fails on the following conditions:

- If the device is actively grabbed by some other client, it returns `AlreadyGrabbed` .

- If grab_window is not viewable, it returns `GrabNotViewable` .

- If the specified time is earlier than the last-grab-time for the specified device or later than the current X server time, it returns `GrabInvalidTime` . Otherwise, the last-grab-time for the specified device is set to the specified time and `CurrentTime` is replaced by the current X server time.

- If the device is frozen by an active grab of another client, it returns `GrabFrozen` .

If a grabbed device is closed by a client while an active grab by that client is in effect, that active grab will be released. Any passive grabs established by that client will be 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.

To release a grab of an extension device, use the `XUngrabDevice` function.

int **XUngrabDevice**(**display*, **device*, *time*);

display	Specifies the connection to the X server.
device	Specifies the desired device.
time	Specifies the time. This may be either a timestamp expressed in milliseconds, or `CurrentTime` .

`XUngrabDevice` allows a client to release an extension input device and any queued events if this client has it grabbed from either `XGrabDevice` or `XGrabDeviceKey` . If any other devices are frozen by the grab, `XUngrabDevice` thaws them. This function 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 `XUngrabDevice` if the event window for an active device grab becomes not viewable or if the client terminates without releasing the grab.

`XUngrabDevice` can generate `BadDevice` errors.

Passively Grabbing a Key

To passively grab a single key on an extension device, use `XGrabDeviceKey` . That device must have previously been opened using the `XOpenDevice` function, or the request will fail with a `BadDevice` error. If the specified device does not support input class `Keys` , the request will fail with a `BadMatch` error.

int **XGrabDeviceKey**(**display*, **device*, *keycode*, *modifiers*, **modifier_device*, *grab_window*, *owner_events*, *event_count*, **event_list*, *this_device_mode*, *other_device_mode*);

display	Specifies the connection to the X server.
device	Specifies the desired device.
keycode	Specifies the keycode of the key that is to be grabbed. You can pass either the keycode 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 key request for all possible modifier combinations (including the combination of no modifiers).

modifier_device	Specifies the device whose modifiers are to be used. If NULL is specified, the core X keyboard is used as the modifier_device.
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_count	Specifies the number of elements in the event_list array.
event_list	Specifies a pointer to a list of event classes that indicate which events the client wishes to receive.
this_device_mode	Controls further processing of events from this device. You can pass one of these constants: `GrabModeSync` or `GrabModeAsync` .
other_device_mode	Controls further processing of events from all other devices. You can pass one of these constants: `GrabModeSync` or `GrabModeAsync` .

`XGrabDeviceKey` is analogous to the core `XGrabKey` function. It creates an explicit passive grab for a key on an extension device. The `XGrabDeviceKey` function establishes a passive grab on a device. Consequently, in the future,

- IF the device is not grabbed and the specified key, which itself can be a modifier key, is logically pressed when the specified modifier keys logically are down on the specified modifier device (and no other keys are down),
- AND no other modifier keys logically are down,
- AND EITHER the grab window is an ancestor of (or is) the focus window or the grab window is a descendent of the focus window and contains the pointer,
- AND a passive grab on the same device and key combination does not exist on any ancestor of the grab window,
- THEN the device is actively grabbed, as for `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.

The interpretation of the remaining arguments is as for `XGrabDevice` . The active grab is terminated automatically when the logical state of the device has the specified key released (independent of the logical state of the modifier keys).

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

A modifier of `AnyModifier` is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned keycodes. A key of `AnyKey` is equivalent to issuing the request for all possible keycodes. Otherwise, the key must be in the range specified by min_keycode and max_keycode in the information returned by the `XListInputDevices` function. If it is not within that range, `XGrabDeviceKey` generates a `BadValue` error.

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

`XGrabDeviceKey` returns `Success` upon successful completion of the request.

`XGrabDeviceKey` can generate `BadAccess` , `BadClass` , `BadDevice` , `BadMatch` , `BadValue` , and `BadWindow` errors.

To release a passive grab of a single key on an extension device, use `XUngrabDeviceKey` .

int **XUngrabDeviceKey**(**display*, **device*, *keycode*, *modifiers*, **modifier_device*, *ungrab_window*);

display	Specifies the connection to the X server.
device	Specifies the desired device.
keycode	Specifies the keycode of the key that is to be ungrabbed. You can pass either the keycode 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 key request for all possible modifier combinations (including the combination of no modifiers).
modifier_device	Specifies the device whose modifiers are to be used. If NULL is specified, the core X keyboard is used as the modifier_device.
ungrab_window	Specifies the ID of a window associated with the device specified above.

`XUngrabDeviceKey` is analogous to the core `XUngrabKey` function. It releases an explicit passive grab for a key on an extension input device.

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

Passively Grabbing a Button

To establish a passive grab for a single button on an extension device, use `XGrabDeviceButton` . The specified device must have previously been opened using the `XOpenDevice` function, or the request will fail with a `BadDevice` error. If the specified device does not support input class `Buttons` , the request will fail with a `BadMatch` error.

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

display	Specifies the connection to the X server.
device	Specifies the desired device.
button	Specifies the code of the button that is to be grabbed. You can pass either the button or `AnyButton` .
modifiers	Specifies the set of keymasks. This mask is the bitwise inclusive OR of these keymask bits: `ShiftMask` , `LockMask` , `ControlMask` , `Mod1Mask` , `Mod2Mask` , `Mod3Mask` , `Mod4Mask` , 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).
modifier_device	Specifies the device whose modifiers are to be used. If NULL is specified, the core X keyboard is used as the modifier_device.
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_count	Specifies the number of elements in the event_list array.
event_list	Specifies a list of event classes that indicates which device events are to be reported to the client.
this_device_mode	Controls further processing of events from this device. You can pass one of these constants: `GrabModeSync` or `GrabModeAsync` .
other_device_mode	Controls further processing of events from all other devices. You can pass one of these constants: `GrabModeSync` or `GrabModeAsync` .

`XGrabDeviceButton` is analogous to the core `XGrabButton` function. It creates an explicit passive grab for a button on an extension input device. Because the server does not track extension devices, no cursor is specified with this request. For the same reason, there is no confine_to parameter. The device must have previously been opened using the `XOpenDevice` function.

The `XGrabDeviceButton` function establishes a passive grab on a device. Consequently, in the future,

- IF the device is not grabbed and the specified button is logically pressed when the specified modifier keys logically are down (and no other buttons or modifier keys are down),

- AND EITHER the grab window is an ancestor of (or is) the focus window OR the grab window is a descendent of the focus window and contains the pointer,
- AND a passive grab on the same device and button/key combination does not exist on any ancestor of the grab window,
- THEN 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.

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

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

A modifier of `AnyModifier` is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned keycodes. A button of `AnyButton` is equivalent to issuing the request for all possible buttons. Otherwise, it is not required that the specified button be assigned to a physical button.

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

`XGrabDeviceButton` can generate `BadAccess` , `BadClass` , `BadDevice` , `BadMatch` , `BadValue` , and `BadWindow` errors.

To release a passive grab of a button on an extension device, use `XUngrabDeviceButton` .

```
int XUngrabDeviceButton( *display, *device, button, modifiers, *modifier_device, ungrab_window);
```

display	Specifies the connection to the X server.
device	Specifies the desired device.
button	Specifies the code of the button that is to be ungrabbed. You can pass either a 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 key request for all possible modifier combinations (including the combination of no modifiers).
modifier_device	Specifies the device whose modifiers are to be used. If NULL is specified, the core X keyboard is used as the modifier_device.

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

`XUngrabDeviceButton` is analogous to the core `XUngrabButton` function. It releases an explicit passive grab for a button on an extension device. That device must have previously been opened using the `XOpenDevice` function, or a `BadDevice` error will result.

A modifier of `AnyModifier` is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers).

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

Thawing a Device

To allow further events to be processed when a device has been frozen, use `XAllowDeviceEvents` .

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

display	Specifies the connection to the X server.
device	Specifies the desired device.
event_mode	Specifies the event mode. You can pass one of these constants: `AsyncThisDevice` , `SyncThisDevice` , `AsyncOtherDevices` , `ReplayThisDevice` , `AsyncAll` , or `SyncAll` .
time	Specifies the time. This may be either a timestamp expressed in milliseconds, or `CurrentTime` .

`XAllowDeviceEvents` releases some queued events if the client has caused a device to freeze. It has no effect if the specified time is earlier than the last-grab time of the most recent active grab for the client and device, or if the specified time is later than the current X server time. The following describes the processing that occurs depending on what constant you pass to the event_mode argument:

- `AsyncThisDevice`
- If the specified device is frozen by the client, event processing for that continues as usual. If the device is frozen multiple times by the client on behalf of multiple separate grabs, `AsyncThisDevice` thaws for all. `AsyncThisDevice` has no effect if the specified device is not frozen by the client, but the device need not be grabbed by the client.
- `SyncThisDevice`
- If the specified device is frozen and actively grabbed by the client, event processing for that device continues normally until the next key or button event is reported to the client. At this time, the specified device again appears to freeze. However, if the reported event causes the grab to be released, the specified device does not freeze. `SyncThisDevice` has no effect if the specified device is not frozen by the client or is not grabbed by the client.
- `ReplayThisDevice`

- If the specified device is actively grabbed by the client and is frozen as the result of an event having been sent to the client (either from the activation of a `GrabDeviceButton` or from a previous `AllowDeviceEvents` with mode `SyncThisDevice` , but not from a `Grab`), the grab is released and that event is completely reprocessed. This time, however, the request ignores any passive grabs at or above (toward the root) the grab-window of the grab just released. The request has no effect if the specified device is not grabbed by the client or if it is not frozen as the result of an event.

- `AsyncOtherDevices`

- If the remaining devices are frozen by the client, event processing for them continues as usual. If the other devices are frozen multiple times by the client on behalf of multiple separate grabs, `AsyncOtherDevices` ``thaws'' for all. `AsyncOtherDevices` has no effect if the devices are not frozen by the client, but those devices need not be grabbed by the client.

- `SyncAll`

- If all devices are frozen by the client, event processing (for all devices) continues normally until the next button or key event is reported to the client for a grabbed device, at which time the devices again appear to freeze. However, if the reported event causes the grab to be released, then the devices do not freeze (but if any device is still grabbed, then a subsequent event for it will still cause all devices to freeze). `SyncAll` has no effect unless all devices are frozen by the client. If any device is frozen twice by the client on behalf of two separate grabs, `SyncAll` "thaws" for both (but a subsequent freeze for `SyncAll` will freeze each device only once).

- `AsyncAll`

- If all devices are frozen by the client, event processing (for all devices) continues normally. If any device is frozen multiple times by the client on behalf of multiple separate grabs, `AsyncAll` ``thaws ''for all. If any device is frozen twice by the client on behalf of two separate grabs, `AsyncAll` ``thaws'' for both. `AsyncAll` has no effect unless all devices are frozen by the client.

`AsyncThisDevice` , `SyncThisDevice` , and `ReplayThisDevice` have no effect on the processing of events from the remaining devices. `AsyncOtherDevices` has no effect on the processing of events from the specified device. When the event_mode is `SyncAll` or `AsyncAll` , the device parameter is ignored.

It is possible for several grabs of different devices (by the same or different clients) to be active simultaneously. If a device is frozen on behalf of any grab, no event processing is performed for the device. It is possible for a single device to be frozen because of several grabs. In this case, the freeze must be released on behalf of each grab before events can again be processed.

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

Controlling Device Focus

The current focus window for an extension input device can be determined using the `XGetDeviceFocus` function. Extension devices are focused using the `XSetDeviceFocus` function in the same way that the keyboard is focused using the core

`XSetInputFocus` function, except that a device ID is passed as a function parameter. One additional focus state, `FollowKeyboard` , is provided for extension devices.

To get the current focus state, revert state, and focus time of an extension device, use `XGetDeviceFocus` .

`int` **`XGetDeviceFocus`**`(` *`*display`*`,` *`*device`*`,` *`*focus_return`*`,` *`*revert_to_return`*`,` *`*focus_time_return`*`);`

display	Specifies the connection to the X server.
device	Specifies the desired device.
focus_return	Specifies the address of a variable into which the server can return the ID of the window that contains the device focus or one of the constants `None` , `PointerRoot` , or `FollowKeyboard` .
revert_to_return	Specifies the address of a variable into which the server can return the current revert_to status for the device.
focus_time_return	Specifies the address of a variable into which the server can return the focus time last set for the device.

`XGetDeviceFocus` returns the focus state, the revert-to state, and the last-focus-time for an extension input device.

`XGetDeviceFocus` can generate `BadDevice` and `BadMatch` errors.

To set the focus of an extension device, use `XSetDeviceFocus` .

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

display	Specifies the connection to the X server.
device	Specifies the desired device.
focus	Specifies the ID of the window to which the device's focus should be set. This may be a window ID, or `PointerRoot` , `FollowKeyboard` , or `None` .
revert_to	Specifies to which window the focus of the device should revert if the focus window becomes not viewable. One of the following constants may be passed: `RevertToParent` , `RevertToPointerRoot` , `RevertToNone` , or `RevertToFollowKeyboard` .
time	Specifies the time. You can pass either a timestamp, expressed in milliseconds, or `CurrentTime` .

`XSetDeviceFocus` changes the focus for an extension input device and the last-focus-change-time. It has no effect if the specified time is earlier than the 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. This function causes the X server to generate `DeviceFocusIn` and `DeviceFocusOut` events.

The action taken by the server when this function is requested depends on the value of the focus argument:

- If the focus argument is `None` , all input events from this device will be discarded until a new focus window is set. In this case, the revert_to argument is ignored.
- If the focus argument is a window ID, it becomes the focus window of the device. If an input event from the device would normally be reported to this window or to one of its inferiors, the event is reported normally. Otherwise, the event is reported relative to the focus window.
- If the focus argument is `PointerRoot` , the focus window is dynamically taken to be the root window of whatever screen the pointer is on at each input event. In this case, the revert_to argument is ignored.
- If the focus argument is `FollowKeyboard` , the focus window is dynamically taken to be the same as the focus of the X keyboard at each input event.

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

- If the revert_to argument 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 the revert_to argument is `RevertToPointerRoot` , `RevertToFollowKeyboard` , or `RevertToNone` , the focus reverts to that value.

When the focus reverts, the X server generates `DeviceFocusIn` and `DeviceFocusOut` events, but the last-focus-change time is not affected.

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

Controlling Device Feedback

To determine the current feedback settings of an extension input device, use `XGetFeedbackControl` .

`XFeedbackState * **XGetFeedbackControl**( *display, *device, *num_feedbacks_return);`

display	Specifies the connection to the X server.
device	Specifies the desired device.
num_feedbacks_return	Returns the number of feedbacks supported by the device.

`XGetFeedbackControl` returns a list of `FeedbackState` structures that describe the feedbacks supported by the specified device. There is an `XFeedbackState` structure for each class of feedback. These are of variable length, but the first three members are common to all.

```
typedef struct {
     XID class;
     int length;
     XID id;
} XFeedbackState;
```

The common members are as follows:

- The class member identifies the class of feedback. It may be compared to constants defined in the file `< X11/extensions/XI.h >`. Currently defined feedback constants include: `KbdFeedbackClass` , `PtrFeedbackClass` , `StringFeedbackClass` , `IntegerFeedbackClass` , `LedFeedbackClass` , and `BellFeedbackClass` .
- The length member specifies the length of the `FeedbackState` structure and can be used by clients to traverse the list.
- The id member uniquely identifies a feedback for a given device and class. This allows a device to support more than one feedback of the same class. Other feedbacks of other classes or devices may have the same ID.

Those feedbacks equivalent to those supported by the core keyboard are reported in class `KbdFeedback` using the `XKbdFeedbackState` structure, which is defined as follows:

```
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 additional members of the `XKbdFeedbackState` structure report the current state of the feedback:

- The click member specifies the key-click volume and has a value in the range 0 (off) to 100 (loud).
- The percent member specifies the bell volume and has a value in the range 0 (off) to 100 (loud).
- The pitch member specifies the bell pitch in Hz. The range of the value is implementation-dependent.

- The duration member specifies the duration in milliseconds of the bell.
- The led_mask member 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 member has a value of `AutoRepeatModeOn` or `AutoRepeatModeOff` .
- The auto_repeats member is a bit vector. Each bit set to 1 indicates that auto-repeat is enabled for the corresponding key. The vector is represented as 32 bytes. Byte N (from 0) contains the bits for keys 8N to 8N + 7, with the least significant bit in the byte representing key 8N.

Those feedbacks equivalent to those supported by the core pointer are reported in class `PtrFeedback` using the `XPtrFeedbackState` structure, which is defined as follows:

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

The additional members of the `XPtrFeedbackState` structure report the current state of the feedback:

- The accelNum member returns the numerator for the acceleration multiplier.
- The accelDenom member returns the denominator for the acceleration multiplier.
- The accelDenom member returns the threshold for the acceleration.

Integer feedbacks are those capable of displaying integer numbers and reported via the `XIntegerFeedbackState` structure. The minimum and maximum values that they can display are reported.

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

The additional members of the `XIntegerFeedbackState` structure report the capabilities of the feedback:

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

String feedbacks are those that can display character information and are reported via the `XStringFeedbackState` structure. Clients set these feedbacks by passing a list of `KeySyms` to be displayed. The `XGetFeedbackControl` function returns the set of key symbols that the feedback can display, as well as the maximum number of symbols that can be displayed. The `XStringFeedbackState` structure is defined as follows:

```
typedef struct {
     XID class;
     int length;
     XID id;
     int max_symbols;
     int  num_syms_supported;
     KeySym *syms_supported;
} XStringFeedbackState;
```

The additional members of the `XStringFeedbackState` structure report the capabilities of the feedback:

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

Bell feedbacks are those that can generate a sound and are reported via the `XBellFeedbackState` structure. 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 members is the same as that of the corresponding fields in the `XKbdFeedbackState` structure.

```
typedef struct {
     XID class;
     int length;
     XID id;
     int percent;
     int pitch;
     int duration;
} XBellFeedbackState;
```

Led feedbacks are those that can generate a light and are reported via the `XLedFeedbackState` structure. Up to 32 lights per feedback are supported. Each bit in led_mask corresponds to one supported light, and the corresponding bit in led_values indicates whether that light is currently on (1) or off (0). 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.

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

`XGetFeedbackControl` can generate `BadDevice` and `BadMatch` errors.

To free the information returned by the `XGetFeedbackControl` function, use `XFreeFeedbackList` .

void **XFreeFeedbackList**(**list*);

list	Specifies the pointer to the `XFeedbackState` structure returned by a previous call to `XGetFeedbackControl` .

`XFreeFeedbackList` frees the list of feedback control information.

To change the settings of a feedback on an extension device, use `XChangeFeedbackControl` . This function modifies the current control values of the specified feedback using information passed in the appropriate `XFeedbackControl` structure for the feedback. Which values are modified depends on the valuemask passed.

int **XChangeFeedbackControl**(**display*, **device*, *valuemask*, **value*);

display	Specifies the connection to the X server.
device	Specifies the desired device.
valuemask	Specifies one value for each bit in the mask (least to most significant bit). The values are associated with the feedbacks for the specified device.
value	Specifies a pointer to the `XFeedbackControl` structure.

`XChangeFeedbackControl` controls the device characteristics described by the `XFeedbackControl` structure. There is an `XFeedbackControl` structure for each class of feedback. These are of variable length, but the first three members are common to all and are as follows:

```
typedef struct {
```

```
     XID class;
     int length;
     XID id;
} XFeedbackControl;
```

Feedback class `KbdFeedback` controls feedbacks equivalent to those provided by the core keyboard using the `KbdFeedbackControl` structure, which is defined as follows:.

```
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;
```

This class controls the device characteristics described by the `XKbdFeedbackControl` structure. These include the key_click_percent, global_auto_repeat, and individual key auto-repeat. Valid modes are `AutoRepeatModeOn` , `AutoRepeatModeOff` , and `AutoRepeatModeDefault` .

Valid masks are as follows:

```
#define   DvKeyClickPercent           (1><<0)
#define   DvPercent                   (1><<0)
#define   DvPitch                     (1><<0)
#define   DvDuration                  (1><<0)
#define   DvLed                       (1><<0)
#define   DvLedMode                   (1><<0)
#define   DvKey                       (1><<0)
#define   DvAutoRepeatMode            (1><<0)
```

Feedback class `PtrFeedback` controls feedbacks equivalent to those provided by the core pointer using the `PtrFeedbackControl` structure, which is defined as follows:

```
typedef struct {
     XID class;
     int length;
     XID id;
     int accelNum;
```

```
     int accelDenom;
     int threshold;
} XPtrFeedbackControl;
```

Which values are modified depends on the valuemask passed.

Valid masks are as follows:

```
#define    DvAccelnum        (1L<<0)
#define    DvAccelDenom      (1L<<1)
#define    DvThreshold       (1L<<2)
```

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. Acceleration takes effect only if the device moves more than threshold pixels at once and applies only to the amount beyond the value in the threshold argument. Setting a value to -1 restores the default. The values of the accelNumerator and threshold fields must be nonzero for the pointer values to be set. Otherwise, the parameters will be unchanged. Negative values generate a `BadValue` error, as does a zero value for the accelDenominator field.

This request fails with a `BadMatch` error if the specified device is not currently reporting relative motion. If a device that is capable of reporting both relative and absolute motion has its mode changed from `Relative` to `Absolute` by an `XSetDeviceMode` request, valuator control values will be ignored by the server while the device is in that mode.

Feedback class `IntegerFeedback` controls integer feedbacks displayed on input devices and are reported via the `IntegerFeedbackControl` structure, which is defined as follows:

```
typedef struct {
     XID class;
     int length;
     XID id;
     int int_to_display;
} XIntegerFeedbackControl;
```

Valid masks are as follows:

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

Feedback class `StringFeedback` controls string feedbacks displayed on input devices and reported via the `StringFeedbackControl` structure, which is defined as follows:

```
typedef struct {
     XID class;
     int length;
     XID id;
     int num_keysyms;
     KeySym *syms_to_display;
} XStringFeedbackControl;
```

Valid masks are as follows:

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

Feedback class `BellFeedback` controls a bell on an input device and is reported via the `BellFeedbackControl` structure, which is defined as follows:

```
typedef struct {
     XID class;
     int length;
     XID id;
     int percent;
     int pitch;
     int duration;
} XBellFeedbackControl;
```

Valid masks are as follows:

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

Feedback class `LedFeedback` controls lights on an input device and are reported via the `LedFeedbackControl` structure, which is defined as follows:

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

Valid masks are as follows:

```
#define   DvLed                       (1L<<4)
```

```
#define    DvLedMode                    (1L<<5)
```

`XChangeFeedbackControl` can generate `BadDevice` , `BadFeedBack` , `BadMatch` , and `BadValue` errors.

Ringing a Bell on an Input Device

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

`int` **`XDeviceBell`**`(` *`*display`*`,` *`*device`*`,` *`feedbackid`*`,` *`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).

`XDeviceBell` 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 - [(base * percent) / 100] + percent
```

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

```
base + [(base * percent) / 100]
```

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

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

Controlling Device Encoding

To get the key mapping of an extension device that supports input class `Keys` , use `XGetDeviceKeyMapping` .

`KeySym *` **`XGetDeviceKeyMapping`**`(` *`*display`*`,` *`*device`*`,` *`first_keycode_wanted`*`,` *`keycode_count`*`,` *`*keysyms_per_keycode_return`*`);`

display	Specifies the connection to the X server.
device	Specifies the desired device.
first_keycode_wanted	Specifies the first keycode that is to be returned.
keycode_count	Specifies the number of keycodes that are to be returned.

keysyms_per_keycode_return	Returns the number of keysyms per keycode.

`XGetDeviceKeyMapping` is analogous to the core `XGetKeyboardMapping` function. It returns the symbols for the specified number of keycodes for the specified extension device.

`XGetDeviceKeyMapping` returns the symbols for the specified number of keycodes for the specified extension device, starting with the specified keycode. The first_keycode_wanted must be greater than or equal to min-keycode as returned by the `XListInputDevices` request (else a `BadValue` error results). The following value:

```
first_keycode_wanted + keycode_count - 1
```

must be less than or equal to max-keycode as returned by the `XListInputDevices` request (else a `BadValue` error results).

The number of elements in the keysyms list is as follows:

```
keycode_count * keysyms_per_keycode_return
```

And KEYSYM number N (counting from zero) for keycode K has an index (counting from zero), in keysyms, of the following:

```
(K - first_keycode_wanted) * keysyms_per_keycode_return + N
```

The keysyms_per_keycode_return value is chosen arbitrarily by the server to be large enough to report all requested symbols. A special KEYSYM value of `NoSymbol` is used to fill in unused elements for individual keycodes.

To free the data returned by this function, use `XFree`.

If the specified device has not first been opened by this client via `XOpenDevice` , this request will fail with a `BadDevice` error. If that device does not support input class `Keys` , this request will fail with a `BadMatch` error.

`XGetDeviceKeyMapping` can generate `BadDevice` , `BadMatch` , and `BadValue` errors.

To change the keyboard mapping of an extension device that supports input class `Keys` , use `XChangeDeviceKeyMapping` .

int **XChangeDeviceKeyMapping**(**display*, **device*, *first_keycode*, *keysyms_per_keycode*, **keysyms*, *num_codes*);

display	Specifies the connection to the X server.
device	Specifies the desired device.
first_keycode	Specifies the first keycode that is to be changed.
keysyms_per_keycode	Specifies the keysyms that are to be used.
keysyms	Specifies a pointer to an array of keysyms.

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

`XChangeDeviceKeyMapping` is analogous to the core `XChangeKeyboardMapping` function. It defines the symbols for the specified number of keycodes for the specified extension keyboard device.

If the specified device has not first been opened by this client via `XOpenDevice` , this request will fail with a `BadDevice` error. If the specified device does not support input class `Keys` , this request will fail with a `BadMatch` error.

The number of elements in the keysyms list must be a multiple of keysyms_per_keycode. Otherwise, `XChangeDeviceKeyMapping` generates a `BadLength` error. The specified first_keycode must be greater than or equal to the min_keycode value returned by the `ListInputDevices` request, or this request will fail with a `BadValue` error. In addition, if the following expression is not less than the max_keycode value returned by the `ListInputDevices` request, the request will fail with a `BadValue` error:

```
        first_keycode + (num_codes / keysyms_per_keycode) - 1
```

`XChangeDeviceKeyMapping` can generate `BadAlloc` , `BadDevice` , `BadMatch` , and `BadValue` errors.

To obtain the keycodes that are used as modifiers on an extension device that supports input class `Keys` , use `XGetDeviceModifierMapping` .

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

display	Specifies the connection to the X server.
device	Specifies the desired device.

`XGetDeviceModifierMapping` is analogous to the core `XGetModifierMapping` function. The `XGetDeviceModifierMapping` function returns a newly created `XModifierKeymap` structure that contains the keys being used as modifiers for the specified device. The structure should be freed after use with `XFreeModifierMapping` . If only zero values appear in the set for any modifier, that modifier is disabled.

`XGetDeviceModifierMapping` can generate `BadDevice` and `BadMatch` errors.

To set which keycodes are to be used as modifiers for an extension device, use `XSetDeviceModifierMapping` .

`int` **`XSetDeviceModifierMapping`**`(` *`*display`*`,` *`*device`*`,` *`*modmap`*`);`

display	Specifies the connection to the X server.
device	Specifies the desired device.
modmap	Specifies a pointer to the `XModifierKeymap` structure.

`XSetDeviceModifierMapping` is analogous to the core `XSetModifierMapping` function. The `XSetDeviceModifierMapping` function specifies the keycodes of the keys, if any, that are to be used as modifiers. A zero value means that no key should be used. No two arguments can have the same nonzero keycode value. Otherwise, `XSetDeviceModifierMapping` generates a `BadValue` error. There are eight modifiers, and the modifiermap member of the `XModifierKeymap` structure contains eight sets of max_keypermod keycodes, one for each modifier in the order `Shift` , `Lock` , `Control` , `Mod1` , `Mod2` , `Mod3` , `Mod4` , and `Mod5` . Only nonzero keycodes have meaning in each set, and zero keycodes are ignored. In addition, all of the nonzero keycodes must be in the range specified by min_keycode and max_keycode reported by the `XListInputDevices` function. Otherwise, `XSetModifierMapping` generates a `BadValue` error. No keycode may appear twice in the entire map. Otherwise, it generates a `BadValue` error.

A X server can impose restrictions on how modifiers can be changed, for example, if certain keys do not generate up transitions in hardware or if multiple modifier keys are not supported. If some such restriction is violated, the status reply is `MappingFailed` , and none of the modifiers are changed. If the new keycodes specified for a modifier differ from those currently defined and any (current or new) keys for that modifier are in the logically down state, the status reply is `MappingBusy` , and none of the modifiers are changed. `XSetModifierMapping` generates a `DeviceMappingNotify` event on a `MappingSuccess` status.

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

Controlling Button Mapping

To set the mapping of the buttons on an extension device, use `XSetDeviceButtonMapping` .

`int` **`XSetDeviceButtonMapping`**`(` *`*display`*`,` *`*device`*`,` *`map[]`*`,` *`nmap`*`);`

display	Specifies the connection to the X server.
device	Specifies the desired device.
map	Specifies the mapping list.
nmap	Specifies the number of items in the mapping list.

`XSetDeviceButtonMapping` sets the mapping of the buttons on an extension device. If it succeeds, the X server generates a `DeviceMappingNotify` event, and `XSetDeviceButtonMapping` returns `MappingSuccess` . Elements of the list are indexed starting from one. The length of the list must be the same as `XGetDeviceButtonMapping` would return, or a `BadValue` error results. The index is a button number, and the element of the list defines the effective number. A zero element disables a button, and elements are not restricted in value by the number of physical buttons. However, no two elements can have the same nonzero value, or a `BadValue` error results. If any of the buttons to be altered are logically in the down state, `XSetDeviceButtonMapping` returns `MappingBusy` , and the mapping is not changed.

`XSetDeviceButtonMapping` can generate `BadDevice` , `BadMatch` , and `BadValue` errors.

To get the button mapping, use `XGetDeviceButtonMapping` .

`int` **`XGetDeviceButtonMapping`**`(` *`*display`*`,` *`*device`*`,` *`map_return[]`*`,` *`nmap`*`);`

display	Specifies the connection to the X server.

device	Specifies the desired device.
map_return	Specifies the mapping list.
nmap	Specifies the number of items in the mapping list.

`XGetDeviceButtonMapping` returns the current mapping of the specified extension device. Elements of the list are indexed starting from one. `XGetDeviceButtonMapping` returns the number of physical buttons actually on the pointer. The nominal mapping for the buttons is the identity mapping: map[i]=i. The nmap argument specifies the length of the array where the button mapping is returned, and only the first nmap elements are returned in map_return.

`XGetDeviceButtonMapping` can generate `BadDevice` and `BadMatch` errors.

Obtaining the State of a Device

To obtain information that describes the state of the keys, buttons, and valuators of an extension device, use `XQueryDeviceState` .

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

display	Specifies the connection to the X server.
device	Specifies the desired device.

`XQueryDeviceState` returns a pointer to an `XDeviceState` structure, which points to a list of structures that describe the state of the keys, buttons, and valuators on the device:

```
typedef struct {
     XID device_id;
     int num_classes;
     XInputClass *data;
} XDeviceState;
```

The structures are of variable length, but the first two members are common to all and are as follows:

```
typedef struct {
     unsigned char class;
     unsigned char length;
} XInputClass;
```

The class member contains a class identifier. This identifier can be compared with constants defined in the file < `X11/extensions/XI.h` >. Currently defined constants are: `KeyClass` , `ButtonClass` , and `ValuatorClass` .

The length member contains the length of the structure and can be used by clients to traverse the list.

The `XValuatorState` structure describes the current state of the valuators on the device. The num_valuators member contains the number of valuators on the device. The mode member is a mask whose bits report the data mode and other state information for the device. The following bits are currently defined:

```
     DeviceMode     1 << 0     Relative = 0, Absolute = 1
     ProximityState     1 << 1     InProximity = 0, OutOfProximity = 1
```

The valuators member contains a pointer to an array of integers that describe the current value of the valuators. If the mode is `Relative` , these values are undefined.

```
typedef struct {
     unsigned char class;
     unsigned char length;
     unsigned char num_valuators;
     unsigned char mode;
     int *valuators;
} XValuatorState;
```

The `XKeyState` structure describes the current state of the keys on the device. Byte N (from 0) contains the bits for key 8N to 8N + 7 with the least significant bit in the byte representing key 8N.

```
typedef struct {
     unsigned char class;
     unsigned char length;
     short num_keys;
     char keys[32];
} XKeyState;
```

The `XButtonState` structure describes the current state of the buttons on the device. Byte N (from 0) contains the bits for button 8N to 8N + 7 with the least significant bit in the byte representing button 8N.

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

`XQueryDeviceState` can generate `BadDevice` errors.

To free the data returned by this function, use `XFreeDeviceState` .

```
void  XFreeDeviceState( *state);
```

state	Specifies the pointer to the `XDeviceState` data returned by a previous call to `XQueryDeviceState` .

`XFreeDeviceState` frees the device state data.

Events

The input extension creates input events analogous to the core input events. These extension input events are generated by manipulating one of the extension input devices. The remainder of this section discusses the following X Input Extension event topics:

- Event types
- Event classes
- Event structures

Event Types

Event types are integer numbers that a client can use to determine what kind of event it has received. The client compares the type field of the event structure with known event types to make this determination.

The core input event types are constants and are defined in the header file `< X11/X.h >`. Extension event types are not constants. Instead, they are dynamically allocated by the extension's request to the X server when the extension is initialized. Because of this, extension event types must be obtained by the client from the server.

The client program determines the event type for an extension event by using the information returned by the `XOpenDevice` request. This type can then be used for comparison with the type field of events received by the client.

Extension events propagate up the window hierarchy in the same manner as core events. If a window is not interested in an extension event, it usually propagates to the closest ancestor that is interested, unless the dont_propagate list prohibits it. Grabs of extension devices may alter the set of windows that receive a particular extension event.

The following table lists the event category and its associated event type or types.

Event Category	Event Type
Device key	`DeviceKeyPress`
	`DeviceKeyRelease`
Device motion	`DeviceButtonPress`
	`DeviceButtonRelease`
	`DeviceMotionNotify`
Device input focus	`DeviceFocusIn`
	`DeviceFocusOut`
Device state notification	`DeviceStateNotify`

Event Category	**Event Type**
Device proximity	`ProximityIn`
	`ProximityOut`
Device mapping	`DeviceMappingNotify`
Device change	`ChangeDeviceNotify`

Event Classes

Event classes are integer numbers that are used in the same way as the core event masks. They are used by a client program to indicate to the server which events that client program wishes to receive.

The core input event masks are constants and are defined in the header file `< X11/X.h >`. Extension event classes are not constants. Instead, they are dynamically allocated by the extension's request to the X server when the extension is initialized. Because of this, extension event classes must be obtained by the client from the server.

The event class for an extension event and device is obtained from information returned by the `XOpenDevice` function. This class can then be used in an `XSelectExtensionEvent` request to ask that events of that type from that device be sent to the client program.

For `DeviceButtonPress` events, the client may specify whether or not an implicit passive grab should be done when the button is pressed. If the client wants to guarantee that it will receive a `DeviceButtonRelease` event for each `DeviceButtonPress` event it receives, it should specify the `DeviceButtonPressGrab` class in addition to the `DeviceButtonPress` class. This restricts the client in that only one client at a time may request `DeviceButtonPress` events from the same device and window if any client specifies this class.

If any client has specified the `DeviceButtonPressGrab` class, any requests by any other client that specify the same device and window and specify either `DeviceButtonPress` or `DeviceButtonPressGrab` will cause an `Access` error to be generated.

If only the `DeviceButtonPress` class is specified, no implicit passive grab will be done when a button is pressed on the device. Multiple clients may use this class to specify the same device and window combination.

The client may also select `DeviceMotion` events only when a button is down. It does this by specifying the event classes `DeviceButton1Motion` through `DeviceButton5Motion` . An input device will support only as many button motion classes as it has buttons.

Event Structures

Each extension event type has a corresponding structure declared in `< X11/extensions/XInput.h >`. All event structures have the following common members:

type	Set to the event type number that uniquely identifies it. For example, when the X server reports a `DeviceKeyPress` event to a client application, it sends an `XDeviceKeyPressEvent` structure.

serial	Set from the serial number reported in the protocol but expanded from the 16-bit least significant bits to a full 32-bit value.
send_event	Set to `True` if the event came from an `XSendEvent` request.
display	Set to a pointer to a structure that defines the display on which the event was read.

Extension event structures report the current position of the X pointer. In addition, if the device reports motion data and is reporting absolute data, the current value of any valuators the device contains is also reported.

Device Key Events

Key events from extension devices contain all the information that is contained in a key event from the X keyboard. In addition, they contain a device ID and report the current value of any valuators on the device, if that device is reporting absolute data. If data for more than six valuators is being reported, more than one key event will be sent. The axes_count member contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the axes_count member and the first axis reported in the current event in the first_axis member. If the device supports input class `Valuators` , but is not reporting absolute mode data, the axes_count member contains zero (0).

The location reported in the x, y and x_root, y_root members is the location of the core X pointer.

The `XDeviceKeyEvent` structure is defined as follows:

```
typedef struct {
     int type;                      /* of event */
     unsigned long serial;          /* # of last request processed */
     Bool send_event;               /* true if from SendEvent request */
     Display *display;              /* Display the event was read from */
     Window window;                 /* "event" window reported relative to */
     XID deviceid;
     Window root;                   /* root window event occurred on */
     Window subwindow;              /* child window */
     Time time;                     /* milliseconds */
     int x, y;                      /* x, y coordinates in event window */
     int x_root;                    /* coordinates relative to root */
     int y_root;                    /* coordinates relative to root */
     unsigned int state;            /* key or button mask */
     unsigned int keycode;          /* detail */
     Bool same_screen;              /* same screen flag */
     unsigned int device_state;     /* device key or button mask */
     unsigned char axes_count;
     unsigned char first_axis;
     int axis_data[6];
```

```
} XDeviceKeyEvent;

typedef XDeviceKeyEvent XDeviceKeyPressedEvent;
typedef XDeviceKeyEvent XDeviceKeyReleasedEvent;
```

Device Button Events

Button events from extension devices contain all the information that is contained in a button event from the X pointer. In addition, they contain a device ID and report the current value of any valuators on the device if that device is reporting absolute data. If data for more than six valuators is being reported, more than one button event may be sent. The axes_count member contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the axes_count member and the first axis reported in the current event in the first_axis member. If the device supports input class `Valuators` , but is not reporting absolute mode data, the axes_count member contains zero (0).

The location reported in the x, y and x_root, y_root members is the location of the core X pointer.

```
typedef struct {
     int type;                     /* of event */
     unsigned long serial;         /* # of last request processed by server */
     Bool send_event;              /* true if from a SendEvent request */
     Display *display;             /* Display the event was read from */
     Window window;                /* "event" window reported relative to */
     XID deviceid;
     Window root;                  /* root window that the event occurred on */
     Window subwindow;             /* child window */
     Time time;                    /* milliseconds */
     int x, y;                     /* x, y coordinates in event window */
     int x_root;                   /* coordinates relative to root */
     int y_root;                   /* coordinates relative to root */
     unsigned int state;           /* key or button mask */
     unsigned int button;          /* detail */
     Bool same_screen;             /* same screen flag */
     unsigned int device_state;    /* device key or button mask */
     unsigned char axes_count;
     unsigned char first_axis;
     int axis_data[6];
} XDeviceButtonEvent;

typedef XDeviceButtonEvent XDeviceButtonPressedEvent;
typedef XDeviceButtonEvent XDeviceButtonReleasedEvent;
```

Device Motion Events

Motion events from extension devices contain all the information that is contained in a motion event from the X pointer. In addition, they contain a device ID and report the current value of any valuators on the device.

The location reported in the x, y and x_root, y_root members is the location of the core X pointer, and so is 2-dimensional.

Extension motion devices may report motion data for a variable number of axes. The axes_count member contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the axes_count member and the first axis reported in the current event in the first_axis member.

```
typedef struct {
     int type;                          /* of event */
     unsigned long serial;              /* # of last request processed by server *
     Bool send_event;                   /* true if from a SendEvent request */
     Display *display;                  /* Display the event was read from */
     Window window;                     /* "event" window reported relative to */
     XID deviceid;
     Window root;                       /* root window that the event occurred on
     Window subwindow;                  /* child window */
     Time time;                         /* milliseconds */
     int x, y;                          /* x, y coordinates in event window */
     int x_root;                        /* coordinates relative to root */
     int y_root;                        /* coordinates relative to root */
     unsigned int state;                /* key or button mask */
     char is_hint;                      /* detail */
     Bool same_screen;                  /* same screen flag */
     unsigned int device_state;         /* device key or button mask */
     unsigned char axes_count;
     unsigned char first_axis;
     int axis_data[6];
} XDeviceMotionEvent;
```

Device Focus Events

These events are equivalent to the core focus events. They contain the same information, with the addition of a device ID to identify which device has had a focus change, and a timestamp.

`DeviceFocusIn` and `DeviceFocusOut` events are generated for focus changes of extension devices in the same manner as core focus events are generated.

```
typedef struct {
     int type;                      /* of event */
     unsigned long serial;          /* # of last request processed by server */
     Bool send_event;               /* true if this came from a SendEvent request
     Display *display;              /* Display the event was read from */
     Window window;                 /* "event" window it is reported relative to *
     XID deviceid;
     int mode;                      /* NotifyNormal, NotifyGrab, NotifyUngrab */
     int detail;
```

```
                                    /*
                                     * NotifyAncestor, NotifyVirtual, NotifyInferi
                                     * NotifyNonLinear,NotifyNonLinearVirtual, Not
                                     * NotifyPointerRoot, NotifyDetailNone
                                     */
     Time time;
} XDeviceFocusChangeEvent;

typedef XDeviceFocusChangeEvent XDeviceFocusInEvent;
typedef XDeviceFocusChangeEvent XDeviceFocusOutEvent;
```

Device StateNotify Event

This event is analogous to the core keymap event but reports the current state of the device for each input class that it supports. It is generated after every `DeviceFocusIn` event and `EnterNotify` event and is delivered to clients who have selected `XDeviceStateNotify` events.

If the device supports input class `Valuators` , the mode member in the `XValuatorStatus` structure is a bitmask that reports the device mode, proximity state, and other state information. The following bits are currently defined:

```
     0x01     Relative = 0, Absolute = 1
     0x02     InProximity = 0, OutOfProximity = 1
```

If the device supports more valuators than can be reported in a single `XEvent` , multiple `XDeviceStateNotify` events will be generated.

```
typedef struct {
     unsigned char class;
     unsigned char length;
} XInputClass;

typedef struct {
     int type;
     unsigned long serial;      /* # of last request processed by server */
     Bool send_event;           /* true if this came from a SendEvent request */
     Display *display;          /* Display the event was read from */
     Window window;
     XID deviceid;
     Time time;
     int num_classes;
     char data[64];
} XDeviceStateNotifyEvent;

typedef struct {
     unsigned char class;
     unsigned char length;
     unsigned char num_valuators;
```

```
     unsigned char mode;
     int valuators[6];
} XValuatorStatus;

typedef struct {
     unsigned char class;
     unsigned char length;
     short num_keys;
     char keys[32];
} XKeyStatus;

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

Device Mapping Event

This event is equivalent to the core `MappingNotify` event. It notifies client programs when the mapping of keys, modifiers, or buttons on an extension device has changed.

```
typedef struct {
     int type;
     unsigned long serial;
     Bool send_event;
     Display *display;
     Window window;
     XID deviceid;
     Time time;
     int request;
     int first_keycode;
     int count;
} XDeviceMappingEvent;
```

ChangeDeviceNotify Event

This event has no equivalent in the core protocol. It notifies client programs when one of the core devices has been changed.

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

```
     Time time;
     int request;
} XChangeDeviceNotifyEvent;
```

Proximity Events

These events have no equivalent in the core protocol. Some input devices such as graphics tablets or touchscreens may send these events to indicate that a stylus has moved into or out of contact with a positional sensing surface.

The event contains the current value of any valuators on the device if that device is reporting absolute data. If data for more than six valuators is being reported, more than one proximity event may be sent. The axes_count member contains the number of axes that are being reported. The server sends as many of these events as are needed to report the device data. Each event contains the total number of axes reported in the axes_count member and the first axis reported in the current event in the first_axis member. If the device supports input class `Valuators` , but is not reporting absolute mode data, the axes_count member contains zero (0).

```
typedef struct {
     int type;                /* ProximityIn or ProximityOut */
     unsigned long serial;    /* # of last request processed by server */
     Bool send_event;         /* true if this came from a SendEvent request */
     Display *display;        /* Display the event was read from */
     Window window;
     XID deviceid;
     Window root;
     Window subwindow;
     Time time;
     int x, y;
     int x_root, y_root;
     unsigned int state;
     Bool same_screen;
     unsigned int device_state;   /* device key or button mask */
     unsigned char axes_count;
     unsigned char first_axis;
     int axis_data[6];
} XProximityNotifyEvent;

typedef XProximityNotifyEvent XProximityInEvent;
typedef XProximityNotifyEvent XProximityOutEvent;
```

Event Handling Functions

This section discusses the X Input Extension event handling functions that allow you to:

- Determine the extension version
- List the available devices
- Enable and disable extension devices

- Change the mode of a device
- Initialize valuators on an input device
- Get input device controls
- Change input device controls
- Select extension device events
- Determine selected device events
- Control event propogation
- Send an event
- Get motion history

Determining the Extension Version

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

display	Specifies the connection to the X server.
name	Specifies the name of the desired extension.

`XGetExtensionVersion` allows a client to determine whether a server supports the desired version of the input extension.

The `XExtensionVersion` structure returns information about the version of the extension supported by the server and is defined as follows:

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

The major and minor versions can be compared with constants defined in the header file < `X11/extensions/XI.h` >. Each version is a superset of the previous versions.

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

Listing Available Devices

A client program that wishes to access a specific device must first determine whether that device is connected to the X server. This is done through the `XListInputDevices` function, which will return a list of all devices that can be opened by the X server. The client program can use one of the names defined in the < `X11/extensions/XI.h` > header file in an `XInternAtom` request to determine the device type of the desired device. This type can then be compared with the device types returned by the `XListInputDevices` request.

`XDeviceInfo * ` **`XListInputDevices`**`( *display, *ndevices);`

display	Specifies the connection to the X server.
ndevices	Specifies the address of a variable into which the server can return the number of input devices available to the X server.

`XListInputDevices` allows a client to determine which devices are available for X input and information about those devices. An array of `XDeviceInfo` structures is returned, with one element in the array for each device. The number of devices is returned in the ndevices argument.

The X pointer device and X keyboard device are reported, as well as all available extension input devices. The use member of the `XDeviceInfo` structure specifies the current use of the device. If the value of this member is `IsXPointer` , the device is the X pointer device. If the value is `IsXKeyboard` , the device is the X keyboard device. If the value is `IsXExtensionDevice` , the device is available for use as an extension input device.

Each `XDeviceInfo` entry contains a pointer to a list of structures that describe the characteristics of each class of input supported by that device. The num_classes member contains the number of entries in that list.

If the device supports input class `Valuators` , one of the structures pointed to by the `XDeviceInfo` structure will be an `XValuatorInfo` structure. The axes member of that structure contains the address of an array of `XAxisInfo` structures. There is one element in this array for each axis of motion reported by the device. The number of elements in this array is contained in the num_axes element of the `XValuatorInfo` structure. The size of the motion buffer for the device is reported in the motion_buffer member of the `XValuatorInfo` structure.

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 structures pointed to by the `XDeviceInfo` structure are defined as follows:

```
typedef struct _XKeyInfo {
     XID class;
     int length;
     unsigned short min_keycode;
     unsigned short max_keycode;
     unsigned short num_keys;
```

```
} XKeyInfo;

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

typedef struct _XValuatorInfo {
     XID class;
     int length;
     unsigned char num_axes;
     unsigned char mode;
     unsigned long motion_buffer;
     XAxisInfoPtr axes;
} XValuatorInfo;
```

The `XAxisInfo` structure pointed to by the `XValuatorInfo` structure is defined as follows:

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

The following atom names are defined in the < `X11/extensions/XI.h` > header file.

```
MOUSE     QUADRATURE
TABLET     SPACEBALL
KEYBOARD     DATAGLOVE
TOUCHSCREEN     EYETRACKER
TOUCHPAD     CURSORKEYS
BUTTONBOX     FOOTMOUSE
BARCODE     ID_MODULE
KNOB_BOX     ONE_KNOB
TRACKBALL     NINE_KNOB\s+1
```

These names can be used in an `XInternAtom` request to return an atom that can be used for comparison with the type member of the `XDeviceInfo` structure.

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

To free the data returned by `XListInputDevices` , use `XFreeDeviceList` .

```
void  XFreeDeviceList( *list);
```

list	Specifies the pointer to the `XDeviceInfo` array returned by a previous call to `XListInputDevices` .

`XFreeDeviceList` frees the list of input device information.

Enabling and Disabling Extension Devices

Each client program that wishes to access an extension device must request that the server open that device by calling the `XOpenDevice` function.

XDevice * **XOpenDevice**(**display*, *device_id*);

display	Specifies the connection to the X server.
device_id	Specifies the ID that uniquely identifies the device to be opened. This ID is obtained from the `XListInputDevices` request.

`XOpenDevice` opens the device for the requesting client and, on success, returns an `XDevice` structure, which is defined as follows:

```
typedef struct {
     XID device_id;
     int num_classes;
     XInputClassInfo *classes;
} XDevice;
```

The `XDevice` structure contains a pointer to an array of `XInputClassInfo` structures. Each element in that array contains information about events of a particular input class supported by the input device.

The `XInputClassInfo` structure is defined as follows:

```
typedef struct {
     unsigned char input_class;
     unsigned char event_type_base;
} XInputClassInfo;
```

A client program can determine the event type and event class for a given event by using macros defined by the input extension. The name of the macro corresponds to the desired event, and the macro is passed the structure that describes the device from which input is desired, for example:

```
     DeviceKeyPress(XDevice *device, event_type, event_class)
```

The macro will fill in the values of the event class to be used in an `XSelectExtensionEvent` request to select the event and the event type to be used in comparing with the event types of events received via `XNextEvent`.

`XOpenDevice` can generate `BadDevice` errors.

Before terminating, the client program should request that the server close the device by calling the `XCloseDevice` function.

`int XCloseDevice( *display, *device);`

display	Specifies the connection to the X server.
device	Specifies the device to be closed.

`XCloseDevice` closes the device for the requesting client and frees the associated `XDevice` structure.

A client may open the same extension device more than once. Requests after the first successful one return an additional `XDevice` structure with the same information as the first, but otherwise have no effect. A single `XCloseDevice` request will terminate that client's access to the device.

Closing a device releases any active or passive grabs the requesting client has established. If the device is frozen only by an active grab of the requesting client, any queued events are released.

If a client program terminates without closing a device, the server will automatically close that device on behalf of the client. This does not affect any other clients that may be accessing that device.

`XCloseDevice` can generate `BadDevice` errors.

Changing the Mode of a Device

Some devices are capable of reporting either relative or absolute motion data. To change the mode of a device from relative to absolute, use `XSetDeviceMode` .

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

display	Specifies the connection to the X server.
device	Specifies the device whose mode should be changed.
mode	Specifies the mode. You can pass `Absolute` or `Relative` .

`XSetDeviceMode` allows a client to request the server to change the mode of a device that is capable of reporting either absolute positional data or relative motion data. If the device is invalid or if the client has not previously requested that the server open the device via an `XOpenDevice` request, this request will fail with a `BadDevice` error. If the device does not support input class `Valuators` or if it is not capable of reporting the specified mode, the request will fail with a `BadMatch` error.

This request will fail and return `DeviceBusy` if another client has already opened the device and requested a different mode.

`XSetDeviceMode` can generate `BadDevice` , `BadMatch` , `BadMode` , and `DeviceBusy` 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 `XSetDeviceValuators` .

`Status` **`XSetDeviceValuators`**`(` *`*display`*`,` *`*device`*`,` *`num_valuators`*`);`

display	Specifies the connection to the X server.
device	Specifies the device whose valuators should be initialized.
valuators	Specifies the values to which each valuator should be set.
first_valuator	Specifies the first valuator to be set.
num_valuators	Specifies the number of valuators to be set.

`XSetDeviceValuators` 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. A `BadValue` error results if the number of valuators supported by the device is less than the following expression:

```
first_valuator + num_valuators
```

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 `BadDevice` , `BadLength` , `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 `XGetDeviceControl` .

`XDeviceControl *` **`XGetDeviceControl`**`(` *`*display`*`,` *`*device`*`,` *`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.

`XGetDeviceControl` 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 members are common to all device controls and are defined as follows:

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

The control may be compared to constants defined in the file `< X11/extensions/XI.h >`. Currently defined device controls include DEVICE_RESOLUTION.

The information returned for the DEVICE_RESOLUTION control is defined in the `XDeviceResolutionState` structure, which is defined as follows:

```
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 zero (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, `XGetDeviceControl` fails with a `BadMatch` error if the specified device has no valuators.

`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 would be done to initialize the device to a known state or configuration. 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. Setting the device control will also fail if the target input device is grabbed by another client or is open 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	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.

`XChangeDeviceControl` 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` is 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` is returned. The information passed varies according to the specified control and is mapped by a structure appropriate for that control. The first two members are common to all device controls:

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

The control may be set using constants defined in the `< X11/extensions/XI.h >` header file. Currently defined device controls include DEVICE_RESOLUTION.

The information that can be changed by the DEVICE_RESOLUTION control is defined in the `XDeviceResolutionControl` structure, which is defined as follows:

```
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 member 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 ),` `XChangeDeviceControl` fails with a `BadValue` error. A `BadValue` error results if the number of valuators supported by the device is less than the following expression:

```
    first_valuator + num_valuators,
```

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

Selecting Extension Device Events

To select device input events, use `XSelectExtensionEvent` . The parameters passed are a pointer to a list of classes that define the desired event types and devices, a count of the number of elements in the list, and the ID of the window from which events are desired.

`int` **`XSelectExtensionEvent`**`(` *`*display`*`,` *`window`*`,` *`*event_list`*`,` *`event_count`*`);`

display	Specifies the connection to the X server.
window	Specifies the ID of the window from which the client wishes to receive events.
event_list	Specifies a pointer to an array of event classes that specify which events are desired.
event_count	Specifies the number of elements in the event_list.

`XSelectExtensionEvent` requests the server to send events that match the events and devices described by the event list and that come from the requested window. The elements of the `XEventClass` array are the event_class values obtained by invoking a macro with the pointer to an `XDevice` structure returned by the `XOpenDevice` request. For example, the `DeviceKeyPress` macro would return the `XEventClass` for `DeviceKeyPress` events from the specified device if it were invoked in the following form:

```
     DeviceKeyPress (XDevice *device, event_type, event_class)
```

Macros are defined for the following event classes:

```
DeviceKeyPress
DeviceKeyRelease
DeviceButtonPress
DeviceButtonRelease
DeviceMotionNotify
DeviceFocusIn
DeviceFocusOut
ProximityIn
ProximityOut
DeviceStateNotify
DeviceMappingNotify
ChangeDeviceNotify
DevicePointerMotionHint
DeviceButton1Motion
DeviceButton2Motion
DeviceButton3Motion,
DeviceButton4Motion
DeviceButton5Motion
DeviceButtonMotion,
DeviceOwnerGrabButton
DeviceButtonPressGrab
```

To get the next available event from within a client program, use the core `XNextEvent` function. This returns the next event whether it came from a core device or an extension device.

Succeeding `XSelectExtensionEvent` requests using event classes for the same device as was specified on a previous request will replace the previous set of selected events from that device with the new set.

`XSelectExtensionEvent` can generate `BadAccess` , `BadClass` , `BadLength` , and `BadWindow` errors.

Determining Selected Device Events

To determine which extension events are currently selected from a given window, use `XGetSelectedExtensionEvents` .

`int` **`XGetSelectedExtensionEvents`**`(` *`*display`*`,` *`window`*`,` *`*this_client_count`*`,` *`**this_client`*`,` *`*all_clients_count`*`,` *`**all_clients`*`);`

display	Specifies the connection to the X server.
window	Specifies the ID of the window from which the client wishes to receive events.
this_client_count	Returns the number of elements in the this_client list.

this_client	Returns a list of `XEventClasses` that specify which events are selected by this client.
all_clients_count	Returns the number of elements in the all_clients list.
all_clients	Returns a list of `XEventClasses` that specify which events are selected by all clients.

`XGetSelectedExtensionEvents` returns pointers to two event class arrays. One lists the extension events selected by this client from the specified window. The other lists the extension events selected by all clients from the specified window. This information is analogous to that returned in your_event_mask and all_event_masks of the `XWindowAttributes` structure when an `XGetWindowAttributes` request is made. To free the two arrays returned by this function, use `XFree`.

`XGetSelectedExtensionEvents` can generate `BadWindow` errors.

Controlling Event Propagation

Extension events propagate up the window hierarchy in the same manner as core events. If a window is not interested in an extension event, it usually propagates to the closest ancestor that is interested, unless the dont_propagate list prohibits it. Grabs of extension devices may alter the set of windows that receive a particular extension event.

Client programs may control event propagation through the use of the following two functions: `XChangeDeviceDontPropagateList` and `XGetDeviceDontPropagateList` .

`int` **`XChangeDeviceDontPropagateList`**`(` *`*display`*`,` *`window`*`,` *`event_count`*`,` *`*events`*`,` *`mode`*`);`

display	Specifies the connection to the X server.
window	Specifies the desired window.
event_count	Specifies the number of elements in the events list.
events	Specifies a pointer to the list of XEventClasses.
mode	Specifies the mode. You can pass `AddToList` or `DeleteFromList` .

`XChangeDeviceDontPropagateList` adds an event to or deletes an event from the do_not_propagate list of extension events for the specified window. There is one list per window, and the list remains for the life of the window. The list is not altered if a client that changed the list terminates.

Suppression of event propagation is not allowed for all events. If a specified `XEventClass` is invalid because suppression of that event is not allowed, a `BadClass` error results.

`XChangeDeviceDontPropagateList` can generate `BadClass` , `BadMode` , and `BadWindow` errors.

`XEventClass *` **`XGetDeviceDontPropagateList`**`(` *`*display`*`,` *`window`*`,` *`*event_count`*`);`

display	Specifies the connection to the X server.
window	Specifies the desired window.
event_count	Returns the number of elements in the array returned by this function.

`XGetDeviceDontPropagateList` allows a client to determine the do_not_propagate list of extension events for the specified window. It returns an array of `XEventClass` , each `XEventClass` representing a device/event type pair. To free the data returned by this function, use `XFree`.

`XGetDeviceDontPropagateList` can generate `BadWindow` errors.

Sending an Event

To send an extension event to another client, use `XSendExtensionEvent` .

```
int  XSendExtensionEvent( *display,  *device,  window,  propagate,
event_count,  *event_list,  *event);
```

display	Specifies the connection to the X server.
device	Specifies the device whose ID is recorded in the event.
window	Specifies the destination window ID. You can pass a window ID, `PointerWindow` or `InputFocus` .
propagate	Specifies a boolean value that is either `True` or `False` .
event_count	Specifies the number of elements in the event_list array.
event_list	Specifies a pointer to an array of `XEventClass` .
event	Specifies a pointer to the event that is to be sent.

`XSendExtensionEvent` identifies the destination window, determines which clients should receive the specified event, and ignores any active grabs. It requires a list of `XEventClass` to be specified. These are obtained by opening an input device with the `XOpenDevice` request.

`XSendExtensionEvent` uses the window argument to identify the destination window as follows:

- If you pass `PointerWindow` , the destination window is the window that contains the pointer.
- If you pass `InputFocus` and if the focus window contains the pointer, the destination window is the window that contains the pointer. If the focus window does not contain the pointer, the destination window is the focus window.

To determine which clients should receive the specified events, `XSendExtensionEvent` uses the propagate argument as follows:

- If propagate is `False` , the event is sent to every client selecting from the destination window any of the events specified in the event_list array.

- If propagate is `True` and no clients have selected from the destination window any of the events specified 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 event 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, 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 sequence number in the event correctly.

`XSendExtensionEvent` returns zero if the conversion-to-wire protocol failed; otherwise, it returns nonzero.

`XSendExtensionEvent` can generate `BadClass` , `BadDevice` , `BadValue` , and `BadWindow` errors.

Getting Motion History

`XDeviceTimeCoord *` **`XGetDeviceMotionEvents`**`(` *`axis_count_return)`*`,` *`*display`*`,` *`*device`*`,` *`stop`*`,` *`*nevents_return`*`,` *`*mode_return`*`,` *`*axis_count_return`*`);`

display	Specifies the connection to the X server.
device	Specifies the desired device.
start	Specifies the start time.
stop	Specifies the stop time.
nevents_return	Returns the number of positions in the motion buffer returned for this request.
mode_return	Returns the mode of the nevents information. The mode will be one of the following: `Absolute` or `Relative` .
axis_count_return	Returns the number of axes reported in each of the positions returned.

`XGetDeviceMotionEvents` returns all positions in the device's motion history buffer that fall between the specified start and stop times inclusive. If the start time is in the future or is later than the stop time, no positions are returned.

The return type for this function is an `XDeviceTimeCoord` structure, which is defined as follows:

```
typedef struct {
```

```
     Time time;
     unsigned int *data;
} XDeviceTimeCoord;
```

The data member is a pointer to an array of data items. Each item is of type int, and there is one data item per axis of motion reported by the device. The number of axes reported by the device is returned in the axis_count variable.

The value of the data items depends on the mode of the device. The mode is returned in the mode variable. If the mode is `Absolute` , the data items are the raw values generated by the device. These may be scaled by the client program using the maximum values that the device can generate for each axis of motion that it reports. The maximum value for each axis is reported in the max_val member of the `XAxisInfo` structure, which is part of the information returned by the `XListInputDevices` request.

If the mode is `Relative` , the data items are the relative values generated by the device. The client program must choose an initial position for the device and maintain a current position by accumulating these relative values.

Consecutive calls to `XGetDeviceMotionEvents` can return data of different modes, that is, if some client program has changed the mode of the device via an `XSetDeviceMode` request.

`XGetDeviceMotionEvents` can generate `BadDevice` and `BadMatch` errors.

To free the data returned by `XGetDeviceMotionEvents` , use `XFreeDeviceMotionEvents` .

`void` **`XFreeDeviceMotionEvents`**`(` *`*events`*`);`

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

`XFreeDeviceMotionEvents` frees the specified array of motion information. `Appendix A`

The following information is contained in the `<X11/extensions/XInput.h>` and `<X11/extensions/XI.h>` header files:

```
/* Definitions used by the library and client */

#ifndef _XINPUT_H_
#define _XINPUT_H_

#ifndef _XLIB_H_
#include <X11/Xlib.h>
#endif

#ifndef _XI_H_
#include "XI.h"
#endif
```

```
#define _deviceKeyPress          0
#define _deviceKeyRelease     1

#define _deviceButtonPress     0
#define _deviceButtonRelease     1

#define _deviceMotionNotify     0

#define _deviceFocusIn          0
#define _deviceFocusOut          1

#define _proximityIn          0
#define _proximityOut          1

#define _deviceStateNotify     0
#define _deviceMappingNotify     1
#define _changeDeviceNotify     2

#define FindTypeAndClass(d, type, class, classid, offset) \
    { int i; XInputClassInfo *ip; \
    type = 0; class = 0; \
    for (i=0, ip= ((XDevice *) d)->classes; \
      i< ((XDevice *) d)->num_classes; \
      i++, ip++) \
     if (ip->input_class == classid) \
         {type =  ip->event_type_base + offset; \
          class =  ((XDevice *) d)->device_id << 8 | type;}}

#define DeviceKeyPress(d, type, class) \
    FindTypeAndClass(d, type, class, KeyClass, _deviceKeyPress)

#define DeviceKeyRelease(d, type, class) \
    FindTypeAndClass(d, type, class, KeyClass, _deviceKeyRelease)

#define DeviceButtonPress(d, type, class) \
    FindTypeAndClass(d, type, class, ButtonClass, _deviceButtonPress)

#define DeviceButtonRelease(d, type, class) \
    FindTypeAndClass(d, type, class, ButtonClass, _deviceButtonRelease)

#define DeviceMotionNotify(d, type, class) \
    FindTypeAndClass(d, type, class, ValuatorClass, _deviceMotionNotify)

#define DeviceFocusIn(d, type, class) \
    FindTypeAndClass(d, type, class, FocusClass, _deviceFocusIn)

#define DeviceFocusOut(d, type, class) \
    FindTypeAndClass(d, type, class, FocusClass, _deviceFocusOut)

#define ProximityIn(d, type, class) \
    FindTypeAndClass(d, type, class, ProximityClass, _proximityIn)

#define ProximityOut(d, type, class) \
```

```
    FindTypeAndClass(d, type, class, ProximityClass, _proximityOut)

#define DeviceStateNotify(d, type, class) \
    FindTypeAndClass(d, type, class, OtherClass, _deviceStateNotify)

#define DeviceMappingNotify(d, type, class) \
    FindTypeAndClass(d, type, class, OtherClass, _deviceMappingNotify)

#define ChangeDeviceNotify(d, type, class) \
    FindTypeAndClass(d, type, class, OtherClass, _changeDeviceNotify)

#define DevicePointerMotionHint(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _devicePointerMotionHint;}

#define DeviceButton1Motion(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceButton1Motion;}

#define DeviceButton2Motion(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceButton2Motion;}

#define DeviceButton3Motion(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceButton3Motion;}

#define DeviceButton4Motion(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceButton4Motion;}

#define DeviceButton5Motion(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceButton5Motion;}

#define DeviceButtonMotion(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceButtonMotion;}

#define DeviceOwnerGrabButton(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceOwnerGrabButton;}

#define DeviceButtonPressGrab(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _deviceButtonGrab;}

#define NoExtensionEvent(d, type, class) \
    { class =  ((XDevice *) d)->device_id << 8 | _noExtensionEvent;}

#define BadDevice(dpy, error) _xibaddevice(dpy, &error)

#define BadClass(dpy, error) _xibadclass(dpy, &error)

#define BadEvent(dpy, error) _xibadevent(dpy, &error)

#define BadMode(dpy, error) _xibadmode(dpy, &error)

#define DeviceBusy(dpy, error) _xidevicebusy(dpy, &error)

/***************************************************************
 *
 * DeviceKey events.  These events are sent by input devices that
```

```
 * support input class Keys.
 * The location of the X pointer is reported in the coordinate
 * fields of the x,y and x_root,y_root fields.
 *
 */

typedef struct
    {
    int             type;          /* of event */
    unsigned long   serial;        /* # of last request processed */
    Bool            send_event;    /* true if from SendEvent request */
    Display         *display;      /* Display the event was read from */
    Window          window;        /* "event" window reported relative to */
    XID             deviceid;
    Window          root;          /* root window event occured on */
    Window          subwindow;     /* child window */
    Time            time;          /* milliseconds */
    int             x, y;          /* x, y coordinates in event window */
    int             x_root;        /* coordinates relative to root */
    int             y_root;        /* coordinates relative to root */
    unsigned int    state;         /* key or button mask */
    unsigned int    keycode;       /* detail */
    Bool            same_screen;   /* same screen flag */
    unsigned int    device_state;  /* device key or button mask */
    unsigned char   axes_count;
    unsigned char   first_axis;
    int             axis_data[6];
    } XDeviceKeyEvent;

typedef XDeviceKeyEvent XDeviceKeyPressedEvent;
typedef XDeviceKeyEvent XDeviceKeyReleasedEvent;

/*******************************************************************
 *
 * DeviceButton events.  These events are sent by extension devices
 * that support input class Buttons.
 *
 */

typedef struct {
    int           type;          /* of event */
    unsigned long serial;        /* # of last request processed by server */
    Bool          send_event;    /* true if from a SendEvent request */
    Display       *display;      /* Display the event was read from */
    Window        window;        /* "event" window reported relative to */
    XID           deviceid;
    Window        root;          /* root window that the event occured on */
    Window        subwindow;     /* child window */
    Time          time;          /* milliseconds */
    int           x, y;          /* x, y coordinates in event window */
    int           x_root;        /* coordinates relative to root */
    int           y_root;        /* coordinates relative to root */
    unsigned int  state;         /* key or button mask */
    unsigned int  button;        /* detail */
```

```
    Bool          same_screen;  /* same screen flag */
    unsigned int  device_state; /* device key or button mask */
    unsigned char axes_count;
    unsigned char first_axis;
    int           axis_data[6];
    } XDeviceButtonEvent;

typedef XDeviceButtonEvent XDeviceButtonPressedEvent;
typedef XDeviceButtonEvent XDeviceButtonReleasedEvent;

/*******************************************************************
 *
 * DeviceMotionNotify event.  These events are sent by extension devices
 * that support input class Valuators.
 *
 */

typedef struct
    {
    int           type;         /* of event */
    unsigned long serial;       /* # of last request processed by server */
    Bool          send_event;   /* true if from a SendEvent request */
    Display       *display;     /* Display the event was read from */
    Window        window;       /* "event" window reported relative to */
    XID           deviceid;
    Window        root;         /* root window that the event occured on */
    Window        subwindow;    /* child window */
    Time          time;         /* milliseconds */
    int           x, y;         /* x, y coordinates in event window */
    int           x_root;       /* coordinates relative to root */
    int           y_root;       /* coordinates relative to root */
    unsigned int  state;        /* key or button mask */
    char          is_hint;      /* detail */
    Bool          same_screen; /* same screen flag */
    unsigned int  device_state; /* device key or button mask */
    unsigned char axes_count;
    unsigned char first_axis;
    int           axis_data[6];
    } XDeviceMotionEvent;

/*******************************************************************
 *
 * DeviceFocusChange events.  These events are sent when the focus
 * of an extension device that can be focused is changed.
 *
 */

typedef struct
    {
    int           type;        /* of event */
    unsigned long serial;      /* # of last request processed by server */
    Bool          send_event; /* true if from a SendEvent request */
    Display       *display;    /* Display the event was read from */
    Window        window;      /* "event" window reported relative to */
```

```
    XID           deviceid;
    int           mode;       /* NotifyNormal, NotifyGrab, NotifyUngrab */
    int           detail;
     /*
      * NotifyAncestor, NotifyVirtual, NotifyInferior,
      * NotifyNonLinear,NotifyNonLinearVirtual, NotifyPointer,
      * NotifyPointerRoot, NotifyDetailNone
      */
    Time                time;
    } XDeviceFocusChangeEvent;

typedef XDeviceFocusChangeEvent XDeviceFocusInEvent;
typedef XDeviceFocusChangeEvent XDeviceFocusOutEvent;

/*******************************************************************
 *
 * ProximityNotify events.  These events are sent by those absolute
 * positioning devices that are capable of generating proximity information.
 *
 */

typedef struct
    {
    int             type;      /* ProximityIn or ProximityOut */
    unsigned long   serial;    /* # of last request processed by server */
    Bool            send_event; /* true if this came from a SendEvent request */
    Display         *display;  /* Display the event was read from */
    Window          window;
    XID                 deviceid;
    Window          root;
    Window          subwindow;
    Time            time;
    int             x, y;
    int             x_root, y_root;
    unsigned int    state;
    Bool            same_screen;
    unsigned int    device_state; /* device key or button mask */
    unsigned char   axes_count;
    unsigned char   first_axis;
    int             axis_data[6];
    } XProximityNotifyEvent;
typedef XProximityNotifyEvent XProximityInEvent;
typedef XProximityNotifyEvent XProximityOutEvent;

/*******************************************************************
 *
 * DeviceStateNotify events are generated on EnterWindow and FocusIn
 * for those clients who have selected DeviceState.
 *
 */

typedef struct
    {
    unsigned char     class;
```

```
    unsigned char     length;
    } XInputClass;

typedef struct {
    int           type;
    unsigned long serial;       /* # of last request processed by server */
    Bool          send_event;   /* true if this came from a SendEvent request */
    Display       *display;     /* Display the event was read from */
    Window        window;
    XID           deviceid;
    Time          time;
    int           num_classes;
    char       data[64];
} XDeviceStateNotifyEvent;

typedef struct {
    unsigned char     class;
    unsigned char     length;
    unsigned char     num_valuators;
    unsigned char     mode;
    int                  valuators[6];
} XValuatorStatus;

typedef struct {
    unsigned char     class;
    unsigned char     length;
    short          num_keys;
    char             keys[32];
} XKeyStatus;

typedef struct {
    unsigned char     class;
    unsigned char     length;
    short          num_buttons;
    char             buttons[32];
} XButtonStatus;

/*******************************************************************
 *
 * DeviceMappingNotify event.  This event is sent when the key mapping,
 * modifier mapping, or button mapping of an extension device is changed.
 *
 */

typedef struct {
    int           type;
    unsigned long serial;       /* # of last request processed by server */
    Bool          send_event;   /* true if this came from a SendEvent request */
    Display       *display;     /* Display the event was read from */
    Window        window;       /* unused */
    XID           deviceid;
    Time          time;
    int           request;      /* one of MappingModifier, MappingKeyboard,
                                    MappingPointer */
```

```
    int           first_keycode;/* first keycode */
    int           count;        /* defines range of change w. first_keycode*/
} XDeviceMappingEvent;

/*******************************************************************
 *
 * ChangeDeviceNotify event.  This event is sent when an
 * XChangeKeyboard or XChangePointer request is made.
 *
 */

typedef struct {
    int           type;
    unsigned long serial;       /* # of last request processed by server */
    Bool          send_event;   /* true if this came from a SendEvent request */
    Display       *display;     /* Display the event was read from */
    Window        window;       /* unused */
    XID           deviceid;
    Time          time;
    int           request;      /* NewPointer or NewKeyboard */
} XChangeDeviceNotifyEvent;

/*******************************************************************
 *
 * Control structures for input devices that support input class
 * Feedback.  These are used by the XGetFeedbackControl and
 * XChangeFeedbackControl functions.
 *
 */

typedef struct {
     XID            class;
     int            length;
     XID            id;
} XFeedbackState;

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;

typedef struct {
    XID     class;
    int     length;
    XID     id;
    int     accelNum;
```

```
    int     accelDenom;
    int     threshold;
} XPtrFeedbackState;

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

typedef struct {
    XID     class;
    int     length;
    XID     id;
    int     max_symbols;
    int     num_syms_supported;
    KeySym  *syms_supported;
} XStringFeedbackState;

typedef struct {
    XID     class;
    int     length;
    XID     id;
    int     percent;
    int     pitch;
    int     duration;
} XBellFeedbackState;

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

typedef struct {
     XID            class;
     int            length;
     XID         id;
} XFeedbackControl;

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

```
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;

typedef struct {
    XID     class;
    int     length;
    XID     id;
    int     num_keysyms;
    KeySym  *syms_to_display;
} XStringFeedbackControl;

typedef struct {
    XID     class;
    int     length;
    XID     id;
    int     int_to_display;
} XIntegerFeedbackControl;

typedef struct {
    XID     class;
    int     length;
    XID     id;
    int     percent;
    int     pitch;
    int     duration;
} XBellFeedbackControl;

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

/*******************************************************************
 *
 * Device control structures.
 *
 */

typedef struct {
     XID            control;
```

```
     int            length;
} XDeviceControl;

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

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

/*******************************************************************
 *
 * An array of XDeviceList structures is returned by the
 * XListInputDevices function.  Each entry contains information
 * about one input device.  Among that information is an array of
 * pointers to structures that describe the characteristics of
 * the input device.
 *
 */

typedef struct _XAnyClassinfo *XAnyClassPtr;

typedef struct _XAnyClassinfo {
    XID      class;
    int      length;
    } XAnyClassInfo;

typedef struct _XDeviceInfo *XDeviceInfoPtr;

typedef struct _XDeviceInfo
    {
    XID                 id;
    Atom                type;
    char                *name;
    int                 num_classes;
    int                 use;
    XAnyClassPtr      inputclassinfo;
    } XDeviceInfo;

typedef struct _XKeyInfo *XKeyInfoPtr;

typedef struct _XKeyInfo
    {
    XID               class;
```

```
    int                 length;
    unsigned short      min_keycode;
    unsigned short      max_keycode;
    unsigned short      num_keys;
    } XKeyInfo;

typedef struct _XButtonInfo *XButtonInfoPtr;

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

typedef struct _XAxisInfo *XAxisInfoPtr;

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

typedef struct _XValuatorInfo *XValuatorInfoPtr;

typedef struct     _XValuatorInfo
    {
    XID                 class;
    int                 length;
    unsigned char       num_axes;
    unsigned char       mode;
    unsigned long       motion_buffer;
    XAxisInfoPtr        axes;
    } XValuatorInfo;


/*******************************************************************
 *
 * An XDevice structure is returned by the XOpenDevice function.
 * It contains an array of pointers to XInputClassInfo structures.
 * Each contains information about a class of input supported by the
 * device, including a pointer to an array of data for each type of event
 * the device reports.
 *
 */


typedef struct {
        unsigned char   input_class;
        unsigned char   event_type_base;
} XInputClassInfo;

typedef struct {
        XID                    device_id;
        int                    num_classes;
```

```
        XInputClassInfo        *classes;
} XDevice;


/*******************************************************************
 *
 * The following structure is used to return information for the
 * XGetSelectedExtensionEvents function.
 *
 */

typedef struct {
        XEventClass     event_type;
        XID             device;
} XEventList;

/*******************************************************************
 *
 * The following structure is used to return motion history data from
 * an input device that supports the input class Valuators.
 * This information is returned by the XGetDeviceMotionEvents function.
 *
 */

typedef struct {
        Time   time;
        int    *data;
} XDeviceTimeCoord;


/*******************************************************************
 *
 * Device state structure.
 * This is returned by the XQueryDeviceState request.
 *
 */

typedef struct {
        XID          device_id;
        int          num_classes;
        XInputClass     *data;
} XDeviceState;

/*******************************************************************
 *
 * Note that the mode field is a bitfield that reports the Proximity
 * status of the device as well as the mode.  The mode field should
 * be OR'd with the mask DeviceMode and compared with the values
 * Absolute and Relative to determine the mode, and should be OR'd
 * with the mask ProximityState and compared with the values InProximity
 * and OutOfProximity to determine the proximity state.
 *
 */
```

```
typedef struct {
    unsigned char     class;
    unsigned char     length;
    unsigned char     num_valuators;
    unsigned char     mode;
    int                  *valuators;
} XValuatorState;

typedef struct {
    unsigned char     class;
    unsigned char     length;
    short          num_keys;
    char             keys[32];
} XKeyState;

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

/*******************************************************************
 *
 * Function definitions.
 *
 */

_XFUNCPROTOBEGIN

extern int     XChangeKeyboardDevice(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */
#endif
);

extern int     XChangePointerDevice(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int               /* xaxis */,
    int               /* yaxis */
#endif
);

extern int     XGrabDevice(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    Window          /* grab_window */,
    Bool          /* ownerEvents */,
    int               /* event count */,
    XEventClass*     /* event_list */,
```

```
    int               /* this_device_mode */,
    int               /* other_devices_mode */,
    Time          /* time */
#endif
);

extern int     XUngrabDevice(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    Time           /* time */
#endif
);

extern int     XGrabDeviceKey(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    unsigned int     /* key */,
    unsigned int     /* modifiers */,
    XDevice*          /* modifier_device */,
    Window          /* grab_window */,
    Bool          /* owner_events */,
    unsigned int     /* event_count */,
    XEventClass*     /* event_list */,
    int               /* this_device_mode */,
    int               /* other_devices_mode */
#endif
);

extern int     XUngrabDeviceKey(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    unsigned int     /* key */,
    unsigned int     /* modifiers */,
    XDevice*          /* modifier_dev */,
    Window          /* grab_window */
#endif
);

extern int     XGrabDeviceButton(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    unsigned int     /* button */,
    unsigned int     /* modifiers */,
    XDevice*          /* modifier_device */,
    Window          /* grab_window */,
    Bool          /* owner_events */,
    unsigned int     /* event_count */,
    XEventClass*     /* event_list */,
    int               /* this_device_mode */,
    int               /* other_devices_mode */
```

```
#endif
);

extern int     XUngrabDeviceButton(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    unsigned int     /* button */,
    unsigned int     /* modifiers */,
    XDevice*          /* modifier_dev */,
    Window          /* grab_window */
#endif
);

extern int     XAllowDeviceEvents(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int               /* event_mode */,
    Time          /* time */
#endif
);

extern int     XGetDeviceFocus(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    Window*          /* focus */,
    int*          /* revert_to */,
    Time*          /* time */
#endif
);

extern int     XSetDeviceFocus(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    Window          /* focus */,
    int               /* revert_to */,
    Time          /* time */
#endif
);

extern XFeedbackState     *XGetFeedbackControl(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int*          /* num_feedbacks */
#endif
);

extern int     XFreeFeedbackList(
#if NeedFunctionPrototypes
    XFeedbackState*     /* list */
```

```
#endif
);

extern int     XChangeFeedbackControl(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    unsigned long     /* mask */,
    XFeedbackControl*     /* f */
#endif
);

extern int     XDeviceBell(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    XID               /* feedbackclass */,
    XID               /* feedbackid */,
    int               /* percent */
#endif
);

extern KeySym     *XGetDeviceKeyMapping(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
#if NeedWidePrototypes
    unsigned int     /* first */,
#else
    KeyCode          /* first */,
#endif
    int               /* keycount */,
    int*          /* syms_per_code */
#endif
);

extern int     XChangeDeviceKeyMapping(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int               /* first */,
    int               /* syms_per_code */,
    KeySym*          /* keysyms */,
    int               /* count */
#endif
);

extern XModifierKeymap     *XGetDeviceModifierMapping(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */
#endif
);
```

```
extern int     XSetDeviceModifierMapping(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    XModifierKeymap*     /* modmap */
#endif
);

extern int     XSetDeviceButtonMapping(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    unsigned char*     /* map[] */,
    int               /* nmap */
#endif
);

extern int     XGetDeviceButtonMapping(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    unsigned char*     /* map[] */,
    unsigned int     /* nmap */
#endif
);

extern XDeviceState     *XQueryDeviceState(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */
#endif
);

extern int     XFreeDeviceState(
#if NeedFunctionPrototypes
    XDeviceState*     /* list */
#endif
);

extern XExtensionVersion *XGetExtensionVersion(
#if NeedFunctionPrototypes
    Display*          /* display */,
    _Xconst char*     /* name */
#endif
);

extern XDeviceInfo     *XListInputDevices(
#if NeedFunctionPrototypes
    Display*          /* display */,
    int*          /* ndevices */
#endif
);

extern int     XFreeDeviceList(
```

```
#if NeedFunctionPrototypes
    XDeviceInfo*     /* list */
#endif
);

extern XDevice     *XOpenDevice(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XID               /* id */
#endif
);

extern int     XCloseDevice(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */
#endif
);

extern int     XSetDeviceMode(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int               /* mode */
#endif
);

extern int     XSetDeviceValuators(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int*          /* valuators */,
    int               /* first_valuator */,
    int               /* num_valuators */
#endif
);

extern XDeviceControl     *XGetDeviceControl(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int               /* control */
#endif
);

extern int     XChangeDeviceControl(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    int               /* control */,
    XDeviceControl*     /* d */
#endif
);
```

```
extern int     XSelectExtensionEvent(
#if NeedFunctionPrototypes
    Display*          /* display */,
    Window          /* w */,
    XEventClass*     /* event_list */,
    int               /* count */
#endif
);

extern int     XGetSelectedExtensionEvents(
#if NeedFunctionPrototypes
    Display*          /* display */,
    Window          /* w */,
    int*          /* this_client_count */,
    XEventClass**     /* this_client_list */,
    int*          /* all_clients_count */,
    XEventClass**     /* all_clients_list */
#endif
);

extern int     XChangeDeviceDontPropagateList(
#if NeedFunctionPrototypes
    Display*          /* display */,
    Window          /* window */,
    int               /* count */,
    XEventClass*     /* events */,
    int               /* mode */
#endif
);

extern XEventClass     *XGetDeviceDontPropagateList(
#if NeedFunctionPrototypes
    Display*          /* display */,
    Window          /* window */,
    int*          /* count */
#endif
);

extern Status     XSendExtensionEvent(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
    Window          /* dest */,
    Bool          /* prop */,
    int               /* count */,
    XEventClass*     /* list */,
    XEvent*          /* event */
#endif
);

extern XDeviceTimeCoord     *XGetDeviceMotionEvents(
#if NeedFunctionPrototypes
    Display*          /* display */,
    XDevice*          /* device */,
```

```
    Time           /* start */,
    Time           /* stop */,
    int*           /* nEvents */,
    int*           /* mode */,
    int*           /* axis_count */
#endif
);

extern int     XFreeDeviceMotionEvents(
#if NeedFunctionPrototypes
    XDeviceTimeCoord*     /* events */
#endif
);

extern int     XFreeDeviceControl(
#if NeedFunctionPrototypes
    XDeviceControl*     /* control */
#endif
);

_XFUNCPROTOEND

#endif /* _XINPUT_H_ */

/* Definitions used by the server, library and client */

#ifndef _XI_H_

#define _XI_H_

#define sz_xGetExtensionVersionReq          8
#define sz_xGetExtensionVersionReply          32
#define sz_xListInputDevicesReq               4
#define sz_xListInputDevicesReply          32
#define sz_xOpenDeviceReq              8
#define sz_xOpenDeviceReply               32
#define sz_xCloseDeviceReq              8
#define sz_xSetDeviceModeReq              8
#define sz_xSetDeviceModeReply              32
#define sz_xSelectExtensionEventReq          12
#define sz_xGetSelectedExtensionEventsReq     8
#define sz_xGetSelectedExtensionEventsReply     32
#define sz_xChangeDeviceDontPropagateListReq     12
#define sz_xGetDeviceDontPropagateListReq     8
#define sz_xGetDeviceDontPropagateListReply     32
#define sz_xGetDeviceMotionEventsReq          16
#define sz_xGetDeviceMotionEventsReply          32
#define sz_xChangeKeyboardDeviceReq          8
#define sz_xChangeKeyboardDeviceReply          32
#define sz_xChangePointerDeviceReq          8
#define sz_xChangePointerDeviceReply          32
#define sz_xGrabDeviceReq              20
#define sz_xGrabDeviceReply               32
#define sz_xUngrabDeviceReq               12
```

```
#define sz_xGrabDeviceKeyReq               20
#define sz_xGrabDeviceKeyReply               32
#define sz_xUngrabDeviceKeyReq               16
#define sz_xGrabDeviceButtonReq               20
#define sz_xGrabDeviceButtonReply          32
#define sz_xUngrabDeviceButtonReq          16
#define sz_xAllowDeviceEventsReq          12
#define sz_xGetDeviceFocusReq               8
#define sz_xGetDeviceFocusReply               32
#define sz_xSetDeviceFocusReq               16
#define sz_xGetFeedbackControlReq          8
#define sz_xGetFeedbackControlReply          32
#define sz_xChangeFeedbackControlReq          12
#define sz_xGetDeviceKeyMappingReq          8
#define sz_xGetDeviceKeyMappingReply          32
#define sz_xChangeDeviceKeyMappingReq          8
#define sz_xGetDeviceModifierMappingReq          8
#define sz_xSetDeviceModifierMappingReq          8
#define sz_xSetDeviceModifierMappingReply     32
#define sz_xGetDeviceButtonMappingReq          8
#define sz_xGetDeviceButtonMappingReply          32
#define sz_xSetDeviceButtonMappingReq          8
#define sz_xSetDeviceButtonMappingReply          32
#define sz_xQueryDeviceStateReq               8
#define sz_xQueryDeviceStateReply          32
#define sz_xSendExtensionEventReq          16
#define sz_xDeviceBellReq               8
#define sz_xSetDeviceValuatorsReq          8
#define sz_xSetDeviceValuatorsReply          32
#define sz_xGetDeviceControlReq               8
#define sz_xGetDeviceControlReply          32
#define sz_xChangeDeviceControlReq          8
#define sz_xChangeDeviceControlReply          32

#define INAME                "XInputExtension"

#define XI_KEYBOARD     "KEYBOARD"
#define XI_MOUSE     "MOUSE"
#define XI_TABLET     "TABLET"
#define XI_TOUCHSCREEN     "TOUCHSCREEN"
#define XI_TOUCHPAD     "TOUCHPAD"
#define XI_BARCODE     "BARCODE"
#define XI_BUTTONBOX     "BUTTONBOX"
#define XI_KNOB_BOX     "KNOB_BOX"
#define XI_ONE_KNOB     "ONE_KNOB"
#define XI_NINE_KNOB     "NINE_KNOB"
#define XI_TRACKBALL     "TRACKBALL"
#define XI_QUADRATURE     "QUADRATURE"
#define XI_ID_MODULE     "ID_MODULE"
#define XI_SPACEBALL     "SPACEBALL"
#define XI_DATAGLOVE     "DATAGLOVE"
#define XI_EYETRACKER     "EYETRACKER"
#define XI_CURSORKEYS     "CURSORKEYS"
#define XI_FOOTMOUSE     "FOOTMOUSE"
```

```
#define Dont_Check               0
#define XInput_Initial_Release          1
#define XInput_Add_XDeviceBell          2
#define XInput_Add_XSetDeviceValuators     3
#define XInput_Add_XChangeDeviceControl     4

#define XI_Absent          0
#define XI_Present          1

#define XI_Initial_Release_Major          1
#define XI_Initial_Release_Minor          0

#define XI_Add_XDeviceBell_Major          1
#define XI_Add_XDeviceBell_Minor          1

#define XI_Add_XSetDeviceValuators_Major     1
#define XI_Add_XSetDeviceValuators_Minor     2

#define XI_Add_XChangeDeviceControl_Major     1
#define XI_Add_XChangeDeviceControl_Minor     3

#define DEVICE_RESOLUTION     1

#define NoSuchExtension          1

#define COUNT               0
#define CREATE               1

#define NewPointer          0
#define NewKeyboard          1

#define XPOINTER          0
#define XKEYBOARD          1

#define UseXKeyboard          0xFF

#define IsXPointer          0
#define IsXKeyboard          1
#define IsXExtensionDevice     2

#define AsyncThisDevice          0
#define SyncThisDevice          1
#define ReplayThisDevice     2
#define AsyncOtherDevices     3
#define AsyncAll          4
#define SyncAll               5

#define FollowKeyboard           3
#define RevertToFollowKeyboard      3

#define DvAccelNum              (1L << 0)
#define DvAccelDenom            (1L << 1)
#define DvThreshold             (1L << 2)
```

```
#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)

#define DvString                (1L << 0)

#define DvInteger               (1L << 0)

#define DeviceMode              (1L << 0)
#define Relative                0
#define Absolute                1

#define ProximityState          (1L << 1)
#define InProximity             (0L << 1)
#define OutOfProximity          (1L << 1)

#define AddToList               0
#define DeleteFromList          1

#define KeyClass            0
#define ButtonClass            1
#define ValuatorClass            2
#define FeedbackClass            3
#define ProximityClass       4
#define FocusClass            5
#define OtherClass            6

#define KbdFeedbackClass       0
#define PtrFeedbackClass       1
#define StringFeedbackClass       2
#define IntegerFeedbackClass       3
#define LedFeedbackClass       4
#define BellFeedbackClass       5

#define _devicePointerMotionHint 0
#define _deviceButton1Motion      1
#define _deviceButton2Motion      2
#define _deviceButton3Motion      3
#define _deviceButton4Motion      4
#define _deviceButton5Motion      5
#define _deviceButtonMotion      6
#define _deviceButtonGrab      7
#define _deviceOwnerGrabButton      8
#define _noExtensionEvent      9

#define XI_BadDevice     0
#define XI_BadEvent     1
#define XI_BadMode     2
```

```
#define XI_DeviceBusy     3
#define XI_BadClass     4

typedef     unsigned long     XEventClass;

/*******************************************************************
 *
 * Extension version structure.
 *
 */

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

#endif /* _XI_H_ */
```

Appendix A. Input Extension Protocol Encoding

`Syntactic Conventions`

All numbers are in decimal, unless prefixed with #x, in which case they are in hexadecimal (base 16).

The general syntax used to describe requests, replies, errors, events, and compound types is:

```
NameofThing
  encode-form
  ...
  encode-form
```

Each encode-form describes a single component.

For components described in the protocol as:

```
name: TYPE
```

the encode-form is:

```
N TYPE name
```

N is the number of bytes occupied in the data stream, and TYPE is the interpretation of those bytes. For example,

```
depth: CARD8
```

becomes:

```
1 CARD8 depth
```

For components with a static numeric value the encode-form is:

```
N value name
```

The value is always interpreted as an N-byte unsigned integer. For example, the first two bytes of a Window error are always zero (indicating an error in general) and three (indicating the Window error in particular):

```
1 0 Error
1 3 code
```

For components described in the protocol as:

```
name: {Name1, ..., NameI}
```

the encode-form is:

```
N  name
 value1 Name1
 ...
 valueI NameI
```

The value is always interpreted as an N-byte unsigned integer. Note that the size of N is sometimes larger than that strictly required to encode the values. For example:

```
class: {InputOutput, InputOnly, CopyFromParent}
```

becomes:

```
2   class
 0 CopyFromParent
 1 InputOutput
 2 InputOnly
```

For components described in the protocol as:

```
NAME: TYPE or Alternative1 ... or AlternativeI
```

the encode-form is:

```
N TYPE   NAME
 value1 Alternative1
 ...
 valueI AlternativeI
```

The alternative values are guaranteed not to conflict with the encoding of TYPE. For example:

```
destination: WINDOW or PointerWindow or InputFocus
```

becomes:

```
4 WINDOW  destination
 0 PointerWindow
 1 InputFocus
```

For components described in the protocol as:

```
value-mask: BITMASK
```

the encode-form is:

```
N BITMASK   value-mask
 mask1 mask-name1
 ...
 maskI mask-nameI
```

The individual bits in the mask are specified and named, and N is 2 or 4. The most-significant bit in a BITMASK is reserved for use in defining chained (multiword) bitmasks, as extensions augment existing core requests. The precise interpretation of this bit is not yet defined here, although a probable mechanism is that a 1-bit indicates that another N bytes of bitmask follows, with bits within the overall mask still interpreted from least-significant to most-significant with an N-byte unit, with N-byte units interpreted in stream order, and with the overall mask being byte-swapped in individual N-byte units.

For LISTofVALUE encodings, the request is followed by a section of the form:

```
VALUEs
 encode-form
 ...
 encode-form
```

listing an encode-form for each VALUE. The NAME in each encode-form keys to the corresponding BITMASK bit. The encoding of a VALUE always occupies four bytes, but the number of bytes specified in the encoding-form indicates how many of the least-significant bytes are actually used; the remaining bytes are unused and their values do not matter.

In various cases, the number of bytes occupied by a component will be specified by a lowercase single-letter variable name instead of a specific numeric value, and often some other component will have its value specified as a simple numeric expression involving these variables. Components specified with such expressions are always interpreted as unsigned integers. The scope of such variables is always just the enclosing request, reply, error, event, or compound type structure. For example:

```
2 3+n   request length
4n LISTofPOINT  points
```

For unused bytes (the values of the bytes are undefined and do not matter), the encode-form is:

```
N   unused
```

If the number of unused bytes is variable, the encode-form typically is:

```
p   unused, p=pad(E)
```

where E is some expression, and pad(E) is the number of bytes needed to round E up to a multiple of four.

```
pad(E) = (4 - (E mod 4)) mod 4
```

`Common Types`

LISTofFOO

- In this document the LISTof notation strictly means some number of repetitions of the FOO encoding; the actual length of the list is encoded elsewhere.

SETofFOO

- A set is always represented by a bitmask, with a 1-bit indicating presence in the set.

BITMASK: CARD32

WINDOW: CARD32

BYTE: 8-bit value

INT8: 8-bit signed integer

INT16: 16-bit signed integer

INT32: 32-bit signed integer

CARD8: 8-bit unsigned integer

CARD16: 16-bit unsigned integer

CARD32: 32-bit unsigned integer

TIMESTAMP: CARD32

EVENTCLASS: CARD32

```
INPUTCLASS
 0 KeyClass
 1 ButtonClass
 2 ValuatorClass
 3 FeedbackClass
 4 ProximityClass
 5 FocusClass
 6 OtherClass
```

```
INPUTCLASS
 0 KbdFeedbackClass
 1 PtrFeedbackClass
 2 StringFeedbackClass
 3 IntegerFeedbackClass
 4 LedFeedbackClass
 5 BellFeedbackClass
```

```
INPUTINFO
 0 KEYINFO
 1 BUTTONINFO
 2 VALUATORINFO
```

```
DEVICEMODE
 0 Relative
 1 Absolute
```

```
PROXIMITYSTATE
 0 InProximity
 1 OutOfProximity
```

```
BOOL
 0 False
 1 True
```

KEYSYM: CARD32

KEYCODE: CARD8

BUTTON: CARD8

```
SETofKEYBUTMASK
 #x0001 Shift
 #x0002 Lock
 #x0004 Control
 #x0008 Mod1
 #x0010 Mod2
 #x0020 Mod3
 #x0040 Mod4
 #x0080 Mod5
 #x0100 Button1
 #x0200 Button2
 #x0400 Button3
 #x0800 Button4
 #x1000 Button5
 #xe000 unused but must be zero
```

```
SETofKEYMASK
 encodings are the same as for SETofKEYBUTMASK, except with
 #xff00  unused but must be zero
```

STRING8: LISTofCARD8

```
STR
 1 n  length of name in bytes
 n STRING8  name
```

```
Errors
```

```
Request
 1 0  Error
 1 1  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
```

```
 1 CARD8  major opcode
 21   unused
```

```
Value
 1 0  Error
 1 2  code
 2 CARD16  sequence number
 4 <32-bits>  bad value
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Window
 1 0  Error
 1 3  code
 2 CARD16  sequence number
 4 CARD32  bad resource id
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Match
 1 0  Error
 1 8  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Access
 1 0  Error
 1 10  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Alloc
```

```
 1 0  Error
 1 11  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Name
 1 0  Error
 1 15  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Device
 1 0  Error
 1 CARD8  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Event
 1 0  Error
 1 CARD8  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Mode
 1 0  Error
 1 CARD8  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

```
Class
 1 0  Error
 1 CARD8  code
 2 CARD16  sequence number
 4   unused
 2 CARD16  minor opcode
 1 CARD8  major opcode
 21   unused
```

Keyboards

KEYCODE values are always greater than 7 (and less than 256).

KEYSYM values with the bit #x10000000 set are reserved as vendor-specific.

The names and encodings of the standard KEYSYM values are contained in appendix F.

Pointers

BUTTON values are numbered starting with one.

Requests

```
GetExtensionVersion
 1 CARD8  input extension opcode
 1 1  GetExtensionVersion opcode
 2 2+(n+p)/4  request length
 2 n  length of name
 2   unused
 n STRING8  name
 p   unused, p=pad(n)
```

```
 =>
 1 1  Reply
 1 1  GetExtensionVersion opcode
 2 CARD16  sequence number
 4 0  reply length
 2 CARD16  major version
 2 CARD16  minor version
 1 BOOL  present
 19   unused
```

```
ListInputDevices
```

```
1 CARD8  input extension opcode
1 2  ListInputDevices opcode
2 1  request length
```

```
=>
1 1   Reply
1 2   ListInputDevices opcode
2 CARD16   sequence number
4 (n+p)/4   reply length
1 CARD8   number of input devices
23    unused
n LISTofDEVICEINFO info for each input device
p    unused, p=pad(n)
```

```
 DEVICEINFO
4 CARD32  device type
1 CARD8  device id
1 CARD8  number of input classes this device reports
1 CARD8  device use
  0 IsXPointer
  1 IsXKeyboard
  2 IsXExtensionDevice
1   unused
n LISTofINPUTINFO   input info for each input class
m STR   name
p    unused, p=pad(m)
```

```
 INPUTINFO KEYINFO or BUTTONINFO or VALUATORINFO
```

```
 KEYINFO
1 0  class id
1 8  length
1 KEYCODE  minimum keycode
1 KEYCODE  maximum keycode
2 CARD16  number of keys
2   unused
```

```
BUTTONINFO
 1 1  class id
 1 4  length
```

```
 2 CARD16  number of buttons

VALUATORINFO
 1 2  class id
 1 8+12n  length
 1 n  number of axes
 1 SETofDEVICEMODE   mode
 4 CARD32  size of motion buffer
 12n LISTofAXISINFO  valuator limits

AXISINFO
 4 CARD32  resolution
 4 CARD32  minimum value
 4 CARD32  maximum value

OpenDevice
 1 CARD8  input extension opcode
 1 3  OpenDevice opcode
 2 2  request length
 1 CARD8  device id
 3   unused

 =>
 1 1   Reply
 1 3   OpenDevice opcode
 2 CARD16   sequence number
 4 (n+p)/4   reply length
 1 CARD8   number of input classes
 23    unused
 n LISTofINPUTCLASSINFO input class information
 p    unused, p=pad(n)

  INPUTCLASSINFO
 1 CARD8  input class id
  0 KEY
  1 BUTTON
  2 VALUATOR
  3 FEEDBACK
  4 PROXIMITY
  5 FOCUS
```

```
  6 OTHER
 1 CARD8  event type base code for this class
```

```
CloseDevice
 1 CARD8  input extension opcode
 1 4  CloseDevice opcode
 2 2  request length
 1 CARD8  device id
 3   unused
```

```
SetDeviceMode
 1 CARD8  input extension opcode
 1 5  SetDeviceMode opcode
 2 2  request length
 1 CARD8  device id
 1 CARD8  mode
 2   unused
```

```
 =>
 1 1   Reply
 1 5   SetDeviceMode opcode
 2 CARD16   sequence number
 4 0   reply length
 1 CARD8   status
  0  Success
   1 AlreadyGrabbed
  3 + first_error DeviceBusy
 23    unused
```

```
SelectExtensionEvent
 1 CARD8   input extension opcode
 1 6   SelectExtensionEvent opcode
 2 3+n    request length
 4 Window   event window
 2 CARD16   count
 2    unused
 4n LISTofEVENTCLASS desired events
```

```
GetSelectedExtensionEvents
 1 CARD8  input extension opcode
```

```
 1 7  GetSelectedExtensionEvents opcode
 2 2  request length
 4 Window  event window
```

```
 =>
 1 1  Reply
 1 7  GetSelecteExtensionEvents opcode
 2 CARD16  sequence number
 4 n + m    reply length
 2 n       this client count
 2 m       all clients count
 20   unused
 4n LISTofEVENTCLASS this client list
 4m LISTofEVENTCLASS all clients list
```

```
ChangeDeviceDontPropagateList
 1 CARD8   input extension opcode
 1 8   ChangeDeviceDontPropagateList opcode
 2 3+n    request length
 4 Window   event window
 2 n   count of events
 1 mode
  0 AddToList
  1 DeleteFromList
 1    unused
 4n LISTofEVENTCLASS desired events
```

```
GetDeviceDontPropagateList
 1 CARD8  input extension opcode
 1 9  GetDeviceDontPropagateList opcode
 2 2  request length
 4 Window  event window
```

```
 =>
 1 1   Reply
 1 9   GetDeviceDontPropagateList opcode
 2 CARD16   sequence number
 4 n     reply length
 2 n   count of events
 22    unused
 4n LISTofEVENTCLASS don't propagate list
```

```
GetDeviceMotionEvents
 1 CARD8  input extension opcode
 1 10  GetDeviceMotionEvents opcode
 2 4  request length
 4 TIMESTAMP  start
  0 CurrentTime
 4 TIMESTAMP  stop
  0 CurrentTime
 1 CARD8  device id
 3   unused
```

```
 =>
 1  1  Reply
 1  10 GetDeviceMotionEvents opcode
 2  CARD16 sequence number
 4  (m+1)n reply length
 4  n  number of DEVICETIMECOORDs in events
 1  m  number of valuators per event
 1  CARD8  mode of the device
   0 Absolute
   1 Relative
 18    unused
 (4m+4)n LISTofDEVICETIMECOORD events
```

```
  DEVICETIMECOORD
 4 TIMESTAMP  time
 4m LISTofINT32  valuators
```

```
ChangeKeyboardDevice
 1 CARD8  input extension opcode
 1 11  ChangeKeyboardDevice opcode
 2 2  request length
 1 CARD8  device id
 3   unused
```

```
 =>
 1 1  Reply
 1 11  ChangeKeyboardDevice opcode
 2 CARD16  sequence number
 4 0  reply length
 1   status
```

```
  0 Success
  1 AlreadyGrabbed
  2 DeviceFrozen
 23   unused
```

```
ChangePointerDevice
 1 CARD8  input extension opcode
 1 12  ChangePointerDevice opcode
 2 2  request length
 1 CARD8  x-axis
 1 CARD8  y-axis
 1 CARD8  device id
 1   unused
```

```
 =>
 1 1  Reply
 1 12  ChangePointerDevice opcode
 2 CARD16  sequence number
 4 0  reply length
 1   status
  0 Success
  1 AlreadyGrabbed
  2 DeviceFrozen
 23   unused
```

```
GrabDevice
 1 CARD8  input extension opcode
 1 13  GrabDevice opcode
 2 5+n   request length
 4 WINDOW  grab-window
 4 TIMESTAMP  time
  0 CurrentTime
 2 n  count of events
 1   this-device-mode
  0 Synchronous
  1 Asynchronous
 1   other-devices-mode
  0 Synchronous
  1 Asynchronous
 1 BOOL  owner-events
 1 CARD8  device id
 2   unused
 4n LISTofEVENTCLASS event list
```

```
 =>
 1 1  Reply
 1 13  GrabDevice opcode
 2 CARD16  sequence number
 4 0  reply length
 1   status
  0 Success
  1 AlreadyGrabbed
  2 InvalidTime
  3 NotViewable
  4 Frozen
 23   unused
```

```
UngrabDevice
 1 CARD8  input extension opcode
 1 14  UngrabDevice opcode
 2 3  request length
 4 TIMESTAMP  time
  0 CurrentTime
 1 CARD8  device id
 3   unused
```

```
GrabDeviceKey
 1 CARD8  input extension opcode
 1 15  GrabDeviceKey opcode
 2 5+n  request length
 4 WINDOW  grab-window
 2 n  count of events
 2 SETofKEYMASK  modifiers
  #x8000 AnyModifier
 1 CARD8  modifier device
  #x0FF UseXKeyboard
 1 CARD8  grabbed device
 1 KEYCODE   key
  0 AnyKey
 1   this-device-mode
  0 Synchronous
  1 Asynchronous
 1   other-devices-mode
  0 Synchronous
  1 Asynchronous
 1 BOOL  owner-events
 2   unused
 4n LISTofEVENTCLASS event list
```

```
UngrabDeviceKey
 1 CARD8  input extension opcode
 1 16  UngrabDeviceKey opcode
 2 4  request length
 4 WINDOW  grab-window
 2 SETofKEYMASK  modifiers
  #x8000 AnyModifier
 1 CARD8  modifier device
  #x0FF UseXKeyboard
 1 KEYCODE   key
  0 AnyKey
 1 CARD8  grabbed device
 3   unused
```

```
GrabDeviceButton
 1 CARD8  input extension opcode
 1 17  GrabDeviceButton opcode
 2 5+n  request length
 4 WINDOW  grab-window
 1 CARD8  grabbed device
 1 CARD8  modifier device
  #x0FF UseXKeyboard
 2 n  count of desired events
 2 SETofKEYMASK  modifiers
 1   this-device-mode
  0 Synchronous
  1 Asynchronous
 1   other-device-mode
  0 Synchronous
  1 Asynchronous
 1 BUTTON  button
  0 AnyButton
 1 BOOL  owner-events
  #x8000 AnyModifier
 2   unused
 4n LISTofEVENTCLASS event list
```

```
UngrabDeviceButton
 1 CARD8  input extension opcode
 1 18  UngrabDeviceButton opcode
 2 4  request length
 4 WINDOW  grab-window
 2 SETofKEYMASK  modifiers
  #x8000 AnyModifier
 1 CARD8  modifier device
  #x0FF UseXKeyboard
 1 BUTTON  button
```

```
  0 AnyButton
 1 CARD8  grabbed device
 3   unused
```

```
AllowDeviceEvents
 1 CARD8  input extension opcode
 1 19  AllowDeviceEvents opcode
 2 3  request length
 4 TIMESTAMP  time
  0 CurrentTime
 1   mode
  0 AsyncThisDevice
  1 SyncThisDevice
  2 ReplayThisDevice
  3 AsyncOtherDevices
  4 AsyncAll
  5 SyncAll
 1 CARD8  device id
 2   unused
```

```
GetDeviceFocus
 1 CARD8  input extension opcode
 1 20  GetDeviceFocus opcode
 2 2  request length
 1 CARD8  device
 3   unused
```

```
 =>
 1 1  Reply
 1 20  GetDeviceFocus opcode
 2 CARD16  sequence number
 4 0  reply length
 4 WINDOW  focus
  0 None
  1 PointerRoot
  3 FollowKeyboard
 4 TIMESTAMP focus time
 1   revert-to
  0 None
  1 PointerRoot
  2 Parent
  3 FollowKeyboard
 15   unused
```

```
SetDeviceFocus
 1 CARD8  input extension opcode
 1 21  SetDeviceFocus opcode
 2 4  request length
 4 WINDOW  focus
  0 None
  1 PointerRoot
  3 FollowKeyboard
 4 TIMESTAMP  time
  0 CurrentTime
 1   revert-to
  0 None
  1 PointerRoot
  2 Parent
  3 FollowKeyboard
 1 CARD8  device
 2   unused
```

```
GetFeedbackControl
 1 CARD8  input extension opcode
 1 22  GetFeedbackControl opcode
 2 2  request length
 1 CARD8  device id
 3   unused
```

```
 =>
 1 1  Reply
 1 22  GetFeedbackControl opcode
 2 CARD16  sequence number
 4 m/4   reply length
 2 n  number of feedbacks supported
 22   unused
 m LISTofFEEDBACKSTATE feedbacks
```

```
  FEEDBACKSTATE KBDFEEDBACKSTATE, PTRFEEDBACKSTATE, INTEGERFEEDBACKSTATE,
  STRINGFEEDBACKSTATE, BELLFEEDBACKSTATE, or LEDFEEDBACKSTATE
```

```
  KBDFEEDBACKSTATE
 1 0  feedback class id
 1 CARD8  id of this feedback
 2 20  length
 2 CARD16  pitch
```

```
2 CARD16  duration
4 CARD32  led_mask
4 CARD32  led_values
1       global_auto_repeat
 0 Off
 1 On
1 CARD8  click
1 CARD8  percent
1       unused
32 LISTofCARD8 auto_repeats
```

PTRFEEDBACKSTATE

```
1 0  feedback class id
1 CARD8  id of this feedback
2 12  length
2   unused
2 CARD16  acceleration-numerator
2 CARD16  acceleration-denominator
2 CARD16  threshold
```

INTEGERFEEDBACKSTATE

```
1 0  feedback class id
1 CARD8  id of this feedback
2 16  length
4 CARD32  resolution
4 INT32  minimum value
4 INT32  maximum value
```

STRINGFEEDBACKSTATE

```
1 1  feedback class id
1 CARD8  id of this feedback
2 4n+8  length
2 CARD16  max_symbols
2 n        number of keysyms supported
4n LISTofKEYSYM  key symbols supported
```

BELLFEEDBACKSTATE

```
1 1  feedback class id
1 CARD8  id of this feedback
2 12  length
1 CARD8  percent
3   unused
2 CARD16  pitch
```

```
 2 CARD16  duration




  LEDFEEDBACKSTATE
 1 1  feedback class id
 1 CARD8  id of this feedback
 2 12  length
 4 CARD32  led_mask
 4 BITMASK  led_values
  #x0001 On
  #x0002 Off




ChangeFeedbackControl
 1 CARD8  input extension opcode
 1 23  ChangeFeedbackControl opcode
 2 3+n/4  request length
 4 BITMASK  value-mask (has n bits set to 1)
  #x0001 keyclick-percent
  #x0002 bell-percent
  #x0004 bell-pitch
  #x0008 bell-duration
  #x0010 led
  #x0020 led-mode
  #x0040 key
  #x0080 auto-repeat-mode
  #x0001 string
  #x0001 integer
  #x0001 acceleration-numerator
  #x0002 acceleration-denominator
  #x0004 acceleration-threshold
 1 CARD8  device id
 1 CARD8  feedback class id
 2   unused
 n FEEDBACKCLASS




  FEEDBACKCLASS KBDFEEDBACKCTL, PTRFEEDBACKCTL, INTEGERFEEDBACKCTL,
  STRINGFEEDBACKCTL, BELLFEEDBACKCTL, or LEDFEEDBACKCTL




  KBDFEEDBACKCTL
 1 0  feedback class id
 1 CARD8  id of this feedback
 2 20  length
 1 KEYCODE  key
 1   auto-repeat-mode
```

```
 0 Off
 1 On
 2 Default
1 INT8  key-click-percent
1 INT8  bell-percent
2 INT16  bell-pitch
2 INT16  bell-duration
4 CARD32  led_mask
4 CARD32  led_values
```

```
 PTRFEEDBACKCTL
1 1  feedback class id
1 CARD8  id of this feedback
2 12  length
2   unused
2 INT16  numerator
2 INT16  denominator
2 INT16  threshold
```

```
 STRINGCTL
1 2  feedback class id
1 CARD8  id of this feedback
2 4n+8  length
2   unused
2 n  number of keysyms to display
4n LISTofKEYSYM  list of key symbols to display
```

```
 INTEGERCTL
1 3  feedback class id
1 CARD8  id of this feedback
2 8  length
4 INT32  integer to display
```

```
 LEDCTL
1 4  feedback class id
1 CARD8  id of this feedback
2 12  length
4 CARD32  led_mask
4 BITMASK  led_values
 #x0001 On
 #x0002 Off
```

```
  BELLCTL
 1 5  feedback class id
 1 CARD8  id of this feedback
 2 8  length
 1 INT8  percent
 3   unused
 2 INT16  pitch
 2 INT16  duration
```

```
GetDeviceKeyMapping
 1 CARD8  input extension opcode
 1 24  GetDeviceKeyMapping opcode
 2 2  request length
 1 CARD8  device
 1 KEYCODE  first-keycode
 1 CARD8  count
 1   unused
```

```
 =>
 1 1  Reply
 1 24  GetDeviceKeyMapping opcode
 2 CARD16  sequence number
 4 nm  reply length (m = count field from the request)
 1 n  keysyms-per-keycode
 23   unused
 4nm LISTofKEYSYM  keysyms
```

```
ChangeDeviceKeyMapping
 1 CARD8  input extension opcode
 1 25  ChangeDeviceKeyMapping opcode
 2 2+nm  request length
 1 CARD8  device
 1 KEYCODE  first-keycode
 1 m  keysyms-per-keycode
 1 n  keycode-count
 4nm LISTofKEYSYM  keysyms
```

```
GetDeviceModifierMapping
 1 CARD8  input extension opcode
 1 26  GetDeviceModifierMapping opcode
```

```
 2 2  request length
 1 CARD8  device
 3   unused




 =>
 1 1  Reply
 1 26  GetDeviceModifierMapping opcode
 2 CARD16  sequence number
 4 2n  reply length
 1 n  keycodes-per-modifier
 23   unused
 8n LISTofKEYCODE  keycodes




SetDeviceModifierMapping
 1 CARD8  input extension opcode
 1 27  SetDeviceModifier opcode
 2 2+2n  request length
 1 CARD8  device
 1 n  keycodes-per-modifier
 2   unused
 8n LISTofKEYCODE  keycodes




 =>
 1 1  Reply
 1 27  SetDeviceModifierMapping opcode
 2 CARD16  sequence number
 4 0  reply length
 1   status
  0 Success
  1 Busy
  2 Failed
 23   unused




GetDeviceButtonMapping
 1 CARD8  input extension opcode
 1 28  GetDeviceButtonMapping opcode
 2 2  request length
 1 CARD8  device
 3   unused
```

```
 =>
 1 1  Reply
 1 28  GetDeviceButtonMapping opcode
 2 CARD16  sequence number
 4 (n+p)/4  reply length
 1 n  number of elements in map list
 23   unused
 n LISTofCARD8   map
 p   unused, p=pad(n)
```

```
SetDeviceButtonMapping
 1 CARD8  input extension opcode
 1 29  SetDeviceButtonMapping opcode
 2 2+(n+p)/4  request length
 1 CARD8  device
 1 n  length of map
 2   unused
 n LISTofCARD8   map
 p   unused, p=pad(n)
```

```
 =>
 1 1  Reply
 1 29  SetDeviceButtonMapping opcode
 2 CARD16  sequence number
 4 0  reply length
 1   status
  0 Success
  1 Busy
 23   unused
```

```
QueryDeviceState
 1 CARD8  input extension opcode
 1 30  QueryDeviceState opcode
 2 2  request length
 1 CARD8  device
 3   unused
```

```
 =>
 1 1   Reply
 1 30   QueryDeviceState opcode
 2 CARD16   sequence number
 4 m/4   reply length
 1 n         number of input classes
```

```
23    unused
m LISTofINPUTSTATE
```

```
 INPUTSTATE KEYSTATE or BUTTONSTATE or VALUATORSTATE
```

```
 KEYSTATE
1 CARD8  key input class id
1 36  length
1 CARD8  num_keys
1   unused
32 LISTofCARD8  status of keys
```

```
 BUTTONSTATE
1 CARD8  button input class id
1 36  length
1 CARD8  num_buttons
1   unused
32 LISTofCARD8   status of buttons
```

```
 VALUATORSTATE
1 CARD8  valuator input class id
1 4n + 4  length
1 n  number of valuators
1   mode
 #x01 DeviceMode (0 = Relative, 1 = Absolute)
 #x02 ProximityState (0 = InProximity, 1 = OutOfProximity)
4n LISTofCARD32  status of valuators
```

```
SendExtensionEvent
 1 CARD8   input extension opcode
 1 31   SendExtensionEvent opcode
 2 4 + 8n + m  request length
 4 WINDOW   destination
 1 CARD8   device
 1 BOOL   propagate
 2 CARD16   eventclass count
 1 CARD8   num_events
 3    unused
 32n LISTofEVENTS  events to send
 4m LISTofEVENTCLASS desired events
```

```
DeviceBell
 1 CARD8  input extension opcode
 1 32  DeviceBell opcode
 2 2  request length
 1 CARD8  device id
 1 CARD8  feedback id
 1 CARD8  feedback class
 1 INT8  percent
```

```
SetDeviceValuators
 1 CARD8  input extension opcode
 1 33  SetDeviceValuators opcode
 2 2 + n  request length
 1 CARD8  device id
 1 CARD8  first valuator
 1 n      number of valuators
 1   unused
 4n LISTofINT32 valuator values to set
```

```
 =>
 1 1   Reply
 1 33   SetDeviceValuators opcode
 2 CARD16   sequence number
 4 0   reply length
 1 CARD8   status
  0 Success
  1 AlreadyGrabbed
 23    unused
```

```
GetDeviceControl
 1 CARD8  input extension opcode
 1 34  GetDeviceControl opcode
 2 2  request length
 2 CARD16  device control type
 1 CARD8  device id
 1   unused
```

```
 =>
 1 1   Reply
 1 34   GetDeviceControl opcode
```

```
 2 CARD16   sequence number
 4 n/4   reply length
 1 CARD8   status
  0   Success
  1   AlreadyGrabbed
  3 + first_error  DeviceBusy
 23    unused
 n    DEVICESTATE
```

```
DEVICESTATE   DEVICERESOLUTIONSTATE
```

```
DEVICERESOLUTIONSTATE
  2 0  control type
  2 8 + 12n  length
  4 n      num_valuators
  4n LISTOfCARD32 resolution values
  4n LISTOfCARD32 resolution min_values
  4n LISTOfCARD32 resolution max_values
```

```
ChangeDeviceControl
  1 CARD8  input extension opcode
  1 35  ChangeDeviceControl opcode
  2 2+n/4  request length
  2 CARD16  control type
  1 CARD8  device id
  1   unused
  n   DEVICECONTROL
```

```
DEVICECONTROL  DEVICERESOLUTIONCTL
```

```
DEVICERESOLUTIONCTL
  2 1  control type
  2 8 + 4n  length
  1 CARD8  first_valuator
  1 n      num_valuators
  2   unused
  4n LISTOfCARD32 resolution values
```

```
 =>
  1 1   Reply
  1 35   ChangeDeviceControl opcode
  2 CARD16   sequence number
  4 0   reply length
  1 CARD8   status
   0   Success
   1   AlreadyGrabbed
   3 + first_error  DeviceBusy
  23    unused
```

Events

DeviceKeyPress, DeviceKeyRelease, DeviceButtonPress, DeviceButtonRelease, ProximityIn, ProximityOut, and DeviceStateNotify events may be followed by zero or more DeviceValuator events. DeviceMotionNotify events will be followed by one or more DeviceValuator events.

```
DeviceValuator
 1 CARD8  code
 1 CARD8  device id
 2 CARD16  sequence number
 2 SETofKEYBUTMASK  state
 1 n  number of valuators this device reports
 1 m  number of first valuator in this event
 24 LISTofINT32 valuators
```

```
DeviceKeyPress
 1 CARD8  code
 1 KEYCODE  detail
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  root
 4 WINDOW  event
 4 WINDOW  child
  0 None
 2 INT16  root-x
 2 INT16  root-y
 2 INT16  event-x
 2 INT16  event-y
 2 SETofKEYBUTMASK  state
 1 BOOL  same-screen
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

```
DeviceKeyRelease
 1 CARD8  code
 1 KEYCODE  detail
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  root
 4 WINDOW  event
 4 WINDOW  child
  0 None
 2 INT16  root-x
 2 INT16  root-y
 2 INT16  event-x
 2 INT16  event-y
 2 SETofKEYBUTMASK   state
 1 BOOL  same-screen
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

```
DeviceButtonPress
 1 CARD8  code
 1 BUTTON  detail
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  root
 4 WINDOW  event
 4 WINDOW  child
  0 None
 2 INT16  root-x
 2 INT16  root-y
 2 INT16  event-x
 2 INT16  event-y
 2 SETofKEYBUTMASK  state
 1 BOOL  same-screen
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

```
DeviceButtonRelease
 1 CARD8  code
 1 BUTTON  detail
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  root
 4 WINDOW  event
 4 WINDOW  child
  0 None
 2 INT16  root-x
 2 INT16  root-y
```

```
 2 INT16  event-x
 2 INT16  event-y
 2 SETofKEYBUTMASK  state
 1 BOOL  same-screen
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

```
DeviceMotionNotify
 1 CARD8  code
 1   detail
  0 Normal
  1 Hint
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  root
 4 WINDOW  event
 4 WINDOW  child
   0 None
 2 INT16  root-x
 2 INT16  root-y
 2 INT16  event-x
 2 INT16  event-y
 2 SETofKEYBUTMASK  state
 1 BOOL  same-screen
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

```
DeviceFocusIn
 1 CARD8  code
 1   detail
  0 Ancestor
  1 Virtual
  2 Inferior
  3 Nonlinear
  4 NonlinearVirtual
  5 Pointer
  6 PointerRoot
  7 None
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  event
 1   mode
  0 Normal
  1 Grab
  2 Ungrab
  3 WhileGrabbed
 1 CARD8  device id
 18   unused
```

```
DeviceFocusOut
 1 CARD8  code
 1   detail
  0 Ancestor
  1 Virtual
  2 Inferior
  3 Nonlinear
  4 NonlinearVirtual
  5 Pointer
  6 PointerRoot
  7 None
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  event
 1   mode
  0 Normal
  1 Grab
  2 Ungrab
  3 WhileGrabbed
 1 CARD8  device id
 18   unused
```

```
ProximityIn
 1 CARD8  code
 1   unused
 2 CARD16  sequence number
 4 TIMESTAMP  time
 4 WINDOW  root
 4 WINDOW  event
 4 WINDOW  child
  0 None
 2 INT16  root-x
 2 INT16  root-y
 2 INT16  event-x
 2 INT16  event-y
 2 SETofKEYBUTMASK   state
 1 BOOL  same-screen
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

```
ProximityOut
 1 CARD8  code
 1   unused
 2 CARD16  sequence number
 4 TIMESTAMP  time
```

```
 4 WINDOW  root
 4 WINDOW  event
 4 WINDOW  child
  0 None
 2 INT16  root-x
 2 INT16  root-y
 2 INT16  event-x
 2 INT16  event-y
 2 SETofKEYBUTMASK   state
 1 BOOL  same-screen
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

DeviceStateNotify events may be immediately followed by zero or one DeviceKeyStateNotify and/ or zero or more DeviceValuator events.

```
DeviceStateNotify
 1 CARD8  code
 1 CARD8  device id
  #x80  MORE_EVENTS follow
 2 CARD16  sequence number
 4 TIMESTAMP  time
 1 CARD8  num_keys
 1 CARD8  num_buttons
 1 CARD8  num_valuators
  1 CARD8  valuator mode and input classes reported
   #x01 reporting keys
   #x02 reporting buttons
   #x04 reporting valuators
   #x40 device mode (0 = Relative, 1 = Absolute)
   #x80 proximity state (0 = InProximity, 1 = OutOfProximity)
 4 LISTofCARD8  first 32 keys (if reported)
 4 LISTofCARD8  first 32 buttons (if reported)
 12 LISTofCARD32  first 3 valuators (if reported)
```

```
DeviceKeyStateNotify
 1 CARD8  code
 1 CARD8  device id
  #x80  MORE_EVENTS follow
 2 CARD16  sequence number
 28 LISTofCARD8  state of keys 33-255
```

```
DeviceButtonStateNotify
 1 CARD8  code
 1 CARD8  device id
  #x80  MORE_EVENTS follow
```

```
 2 CARD16  sequence number
 28 LISTofCARD8  state of buttons 33-255
```

```
DeviceValuator
 1 CARD8  code
 1 CARD8  device id
 2 CARD16  sequence number
 2 SETofKEYBUTMASK  state
 1 n  number of valuators this device reports
 1 n  number of first valuator in this event
 24 LISTofINT32 valuators
```

```
DeviceMappingNotify
 1 CARD8  code
 1 CARD8  device id
 2 CARD16  sequence number
 1   request
  0 MappingModifier
  1 MappingKeyboard
  2 MappingPointer
 1 KEYCODE  first-keycode
 1 CARD8  count
 1   unused
 4 TIMESTAMP  time
 20   unused
```

```
ChangeDeviceNotify
 1 CARD8  code
 1 CARD8  id of device specified on change request
 2 CARD16  sequence number
 4 TIMESTAMP  time
 1   request
  0 NewPointer
  1 NewKeyboard
 23   unused
```