X11 Input Extension Protocol Specification

Version 1.0

MIT X Consortium Standard

X Version 11, Release 5

Mark Patrick Ardent Computer

George Sachs Hewlett-Packard

1.1. Input Extension Overview

This document defines an extension to the X11 protocol to support input devices other than the core X keyboard and pointer. An accompanying document defines a corresponding extension to Xlib (similar extensions for languages other than C are anticipated). This first section gives an overview of the input extension. The next section defines the new protocol requests defined by the extension. We conclude with a description of the new input events generated by the additional input devices.

1.2. Design Approach

The design approach of the extension is to define requests and events analogous to the core requests and events. This allows extension input devices to be individually distinguishable from each other and from the core input devices. These requests 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 reportable via the core input events.

1.3. 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. Requests are defined that allow client programs to change which physical devices are used as the core devices.

1.4. 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.

A client that wishes to access a specific device must first determine whether that device is connected to the X server. This is done through the **ListInputDevices** request, which will return a list of all devices that can be opened by the X server. A client can then open one or more of these devices using the **OpenDevice** request, specify what events they are interested in receiving, and receive and process input events from extension devices in the same way as events from the X keyboard and X pointer. Input events from these devices are of extension types (**DeviceKeyPress**, **DeviceKeyRelease**, **DeviceButtonPress**, **DeviceButtonRelease**, **DeviceMotionNotify**, etc.) and contain a device identifier so that events of the same type coming from different input devices can be distinguished.

Any kind of input device may be used as an extension input device. Extension input devices may have 0 or more keys, 0 or more buttons, and may report 0 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 classes**. Server implementors may add new classes of input devices without changing the protocol requests. Input classes are unique numbers registered with the X Consortium. Each extension input device may support multiple input classes.

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 events reported by the server to a client are of fixed size (32 bytes). In order to represent the change in state of an input device the extension may need to generate a sequence of input events. A client side library (such as Xlib) will typically take these raw input events and format them into a form more convenient to the client.

1.4.1. Event Classes

In the core protocol a client registers interest in receiving certain input events directed to a window by modifying that window's event-mask. Most of the bits in the event mask are already used to specify interest in core X events. The input extension specifies a different mechanism by which a client can express interest in events generated by this extension.

When a client opens a extension input device via the **OpenDevice** request, an **XDevice** structure is returned. Macros are provided that extract 32-bit numbers called **event classes** from that structure, that a client can use to register interest in extension events via the **SelectExtensionEvent** request. The event class combines the desired event type and device id, and may be thought of as the equivalent of core event masks.

1.4.2. Input 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 requests. 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 and reports focus events.

FEEDBACK
The device supports feedbacks.

OTHER
The **ChangeDeviceNotify**, **DeviceMappingNotify**, and **DeviceStateNotify** macros may be invoked passing the **XDevice** structure returned for this device.

Each extension input device may support multiple input classes. Additional classes may be added in the future. Requests that support multiple input classes, such as the **ListInputDevices** function that lists all available input devices, organize the data they return by input class. Client programs that use these requests 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 requests to be recompiled.

2. Requests

Extension input devices are accessed by client programs through the use of new protocol requests. This section summarizes the new requests defined by this extension. The syntax and type definitions used below follow the notation used for the X11 core protocol.

2.1. Getting the Extension Version

The **GetExtensionVersion** request returns version information about the input extension.

```
GetExtensionVersion
        name: STRING
=>
        present: BOOL
        protocol-major-version: CARD16
        protocol-minor-version: CARD16
```

The protocol version numbers returned indicate the version of the input extension supported by the target X server. The version numbers can be compared to constants defined in the header file **XI.h**. Each version is a superset of the previous versions.

2.2. Listing Available Devices

A client that wishes to access a specific device must first determine whether that device is connected to the X server. This is done through the **ListInputDevices** request, which will return a list of all devices that can be opened by the X server.

```
        ListInputDevices
=>
        input-devices: LISTofDEVICEINFO
```

where

```
        DEVICEINFO:        [type: ATOM
                            id: CARD8
                            num_classes: CARD8
                            use: {IsXKeyboard, IsXPointer, IsExtensionDevice}
                            info: LISTofINPUTINFO
                            name: STRING8]

        INPUTINFO:         {KEYINFO, BUTTONINFO, VALUATORINFO}

        KEYINFO:           [class: CARD8
                            length: CARD8
                            min-keycode: KEYCODE
                            max-keycode: KEYCODE
                            num-keys: CARD16]

        BUTTONINFO:        [class: CARD8
                            length: CARD8
                            num-buttons: CARD16]
```

VALUATORINFO: [class: CARD8
length: CARD8
num_axes: CARD8
mode: SETofDEVICEMODE
motion_buffer_size: CARD32
axes: LISTofAXISINFO]

AXISINFO: [resolution: CARD32
min-val: CARD32
max-val: CARD32]

DEVICEMODE: {Absolute, Relative}

Errors: None

This request returns a list of all devices that can be opened by the X server, including the core X keyboard and X pointer. Some implementations may open all input devices as part of X initialization, while others may not open an input device until requested to do so by a client program.

- The information returned for each device is as follows:

The **type** field is of type **Atom** and indicates the nature of the device. Clients may determine device types by invoking the **XInternAtom** request passing one of the names defined in the header file **XI.h**. The following names have been defined to date:

MOUSE
TABLET
KEYBOARD
TOUCHSCREEN
TOUCHPAD
BUTTONBOX
BARCODE
KNOB_BOX
TRACKBALL
QUADRATURE
SPACEBALL
DATAGLOVE
EYETRACKER
CURSORKEYS
FOOTMOUSE
ID_MODULE
ONE_KNOB
NINE_KNOB

The **id** is a small cardinal value in the range 0-128 that uniquely identifies the device. It is assigned to the device when it is initialized by the server. Some implementations may not open an input device until requested by a client program, and may close the device when the last client accessing it requests that it be closed. If a device is opened by a client program via **XOpenDevice**, then closed via **XCloseDevice**, then opened again, it is not guaranteed to have the same id after the second open request.

The **num_classes** field is a small cardinal value in the range 0-255 that specifies the number of input classes supported by the device for which information is returned by **ListInputDevices**. Some input classes, such as class **Focus** and class **Proximity** do not have any information to be returned by **ListInputDevices**.

The **use** field specifies how the device is currently being used. If the value is **IsXKeyboard**, the device is currently being used as the X keyboard. If the value is **IsXPointer**, the device is currently being used as the X pointer. If the value is **IsXExtensionDevice**, the device is available for use as an extension device.

The **name** field contains a pointer to a null-terminated string that corresponds to one of the defined device types.

- **InputInfo** is one of: **KeyInfo**, **ButtonInfo** or **ValuatorInfo**. The first two fields are common to all three:

The **class** field is a cardinal value in the range 0-255. It uniquely identifies the class of input for which information is returned.

The **length** field is a cardinal value in the range 0-255. It specifies the number of bytes of data that are contained in this input class. The length includes the class and length fields.

The remaining information returned for input class **KEYCLASS** is as follows:

min_keycode is of type KEYCODE. It specifies the minimum keycode that the device will report. The minimum keycode will not be smaller than 8.

max_keycode is of type KEYCODE. It specifies the maximum keycode that the device will report. The maximum keycode will not be larger than 255.

num_keys is a cardinal value that specifies the number of keys that the device has.

The remaining information returned for input class **BUTTONCLASS** is as follows:

num_buttons is a cardinal value that specifies the number of buttons that the device has.

The remaining information returned for input class **VALUATORCLASS** is as follows:

mode is a constant that has one of the following values: **Absolute** or **Relative**. Some devices allow the mode to be changed dynamically via the **SetDeviceMode** request.

motion_buffer_size is a cardinal number that specifies the number of elements that can be contained in the motion history buffer for the device.

The **axes** field contains a pointer to an AXISINFO struture.

- The information returned for each axis reported by the device is:

The **resolution** is a cardinal value in counts/meter.

The **min_val** field is a cardinal value in that contains the minimum value the device reports for this axis. For devices whose mode is **Relative**, the min_val field will contain 0.

The **max_val** field is a cardinal value in that contains the maximum value the device reports for this axis. For devices whose mode is **Relative**, the max_val field will contain 0.

2.3. Enabling Devices

Client programs that wish to access an extension device must request that the server open that device. This is done via the **OpenDevice** request.

```
OpenDevice
        id: CARD8
=>

        DEVICE:              [device_id: XID
                              num_classes: INT32
                              classes: LISTofINPUTCLASSINFO]

        INPUTCLASSINFO:      [input_class: CARD8
                              event_type_base: CARD8]
```

Errors: Device

This request returns the event classes to be used by the client to indicate which events the client program wishes to receive. Each input class may report several event classes. For example, input class **Keys** reports **DeviceKeyPress** and **DeviceKeyRelease** event classes. Input classes are unique numbers registered with the X Consortium. Input class **Other** exists to report event classes that are not specific to any one input class, such as **DeviceMappingNotify**, **ChangeDeviceNotify**, and **DeviceStateNotify**.

- The information returned for each device is as follows:

The **device_id** is a number that uniquely identifies the device.

The **num_classes** field contains the number of input classes supported by this device.

- For each class of input supported by the device, the **InputClassInfo** structure contains the following information:

The **input_class** is a small cardinal number that identifies the class of input.

The **event_type_base** is a small cardinal number that specifies the event type of one of the events reported by this input class. This information is not directly used by client programs. Instead, the **Device** is used by macros that return extension event types and event classes. This is described in the section of this document entitled "Selecting Extension Device Events".

Before it exits, the client program should explicitly request that the server close the device. This is done via the **CloseDevice** request.

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 **CloseDevice** 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, the queued events are released when the client terminates.

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.

CloseDevice
    device: DEVICE

    Errors: Device

2.4. 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 the **SetDeviceMode** request. The valid values are **Absolute** or **Relative**.

This request will fail and return **DeviceBusy** if another client already has the device open with a different mode. It will fail and return **AlreadyGrabbed** if another client has the device grabbed. The request will fail with a **BadMatch** error if the requested mode is not supported by the device.

SetDeviceMode
    device: DEVICE
    mode: {Absolute, Relative}

Errors: Device, Match, Mode

=>

status: {Success, DeviceBusy, AlreadyGrabbed}

2.5. Initializing Valuators on an Input Device

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

SetDeviceValuators
device: DEVICE
first_valuator: CARD8
num_valuators: CARD8
valuators: LISTOFINT32

Errors: Length, Device, Match, Value

=>

status: {Success, AlreadyGrabbed}

This request initializes the specified valuators on the specified extension input device. Valuators are numbered beginning with zero. Only the valuators in the range specified by first_valuator and num_valuators are set. If the number of valuators supported by the device is less than the expression first_valuator + num_valuators, a **Value** error will result.

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

2.6. Getting Input Device Controls

GetDeviceControl
device: DEVICE
control: XID

Errors: Length, Device, Match, Value

=>

controlState: {DeviceState}

where

DeviceState: DeviceResolutionState

Errors: Length, Device, Match, Value

This request returns the current state of the specified device control. The device control must be supported by the target server and device or an error will result.

If the request is successful, a pointer to a generic DeviceState structure will be returned. The information returned varies according to the specified control and is mapped by a structure appropriate for that control.

GetDeviceControl will fail with a BadValue error if the server does not support the specified control. It will fail with a BadMatch error if the device does not support the specified control.

Supported device controls and the information returned for them include:

DEVICE_RESOLUTION: [control: CARD16
length: CARD16
num_valuators: CARD8
resolutions: LISTofCARD32
min_resolutions: LISTofCARD32
max_resolutions: LISTofCARD32]

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

When this control is specified, XGetDeviceControl will fail with a BadMatch error if the specified device has no valuators.

ChangeDeviceControl
device: DEVICE
XID: controlId
control: DeviceControl

where

DeviceControl: DeviceResolutionControl

Errors: Length, Device, Match, Value
=>
status: {Success, DeviceBusy, AlreadyGrabbed}

ChangeDeviceControl changes the specifed device control according to the values specified in the DeviceControl structure. The device control must be supported by the target server and device or an error will result.

The information passed with this request varies according to the specified control and is mapped by a structure appropriate for that control.

ChangeDeviceControl will fail with a BadValue error if the server does not support the specified control. It will fail with a BadMatch error if the server supports the specified control, but the requested device does not. The request will fail and return a status of DeviceBusy if another client already has the device open with a device control state that conflicts with the one specified in the request. It will fail with a status of AlreadyGrabbed if some other client has grabbed the specified device. If the request succeeds, Success is returned. If it fails, the device control is left unchanged.

Supported device controls and the information specified for them include:

DEVICE_RESOLUTION: [control: CARD16
length: CARD16
first_valuator: CARD8

num_valuators: CARD8
resolutions: LISTofCARD32]

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. num_valuators specifies the number of valuators in the resolutions list.

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

If the request fails for any reason, none of the valuator resolutions will be changed.

2.7. Selecting Extension Device Events

Extension input events are selected using the **SelectExtensionEvent** request.

SelectExtensionEvent
window: WINDOW
interest: LISTofEVENTCLASS

Errors: Window, Class, Access

This request specifies to the server the events within the specified window which are of interest to the client. As with the core **XSelectInput** function, multiple clients can select input on the same window.

XSelectExtensionEvent requires a list of *event classes*. An event class is a 32-bit number that combines an event type and device id, and is used to indicate which event a client wishes to receive and from which device it wishes to receive it. Macros are provided to obtain event classes from the data returned by the **XOpenDevice** request. The names of these macros correspond to the desired events, i.e. the **DeviceKeyPress** is used to obtain the event class for **DeviceKeyPress** events. The syntax of the macro invocation is:

DeviceKeyPress (device, event_type, event_class);
device: DEVICE
event_type: INT
event_class: INT

The value returned in **event_type** is the value that will be contained in the event type field of the **XDeviceKeyPressEvent** when it is received by the client. The value returned in **event_class** is the value that should be passed in making an **XSelectExtensionEvent** request to receive **DeviceKeyPress** events.

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** event class as well as the **DeviceButtonPress** event 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 **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.

A client may also specify the **DeviceOwnerGrabButton** class. If it has specified both the **DeviceButtonPressGrab** and the **DeviceOwnerGrabButton** classes, implicit passive grabs will activate with owner_events set to **True**. If only the **DeviceButtonPressGrab** class is specified, implicit passive grabs will activate with owner_events set to **False**.

The client may select **DeviceMotion** events only when a button is down. It does this by specifying the event classes **Button1Motion** through **Button5Motion**, or **ButtonMotion**. An input device will only support as many button motion classes as it has buttons.

2.8. Determining Selected Events

To determine which extension events are currently selected from a given window, use **GetSelectedExtensionEvents**.

```
GetSelectedExtensionEvents
        window: WINDOW
=>
        this-client: LISTofEVENTCLASS
        all-clients: LISTofEVENTCLASS

        Errors: Window
```

This request returns two lists specifying the events selected on the specified window. One list gives the extension events selected by this client from the specified window. The other list gives the extension events selected by all clients from the specified window. This information is equivalent to that returned by your-event-mask and all-event-masks in a **GetWindowAttributes** request.

2.9. 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 extension event propagation through the use of the following two requests.

XChangeDeviceDontPropagateList adds an event to or deletes an event from the do_not_propagate list of extension events for the specified window. This list is maintained for the life of the window, and is not altered if the client terminates.

```
ChangeDeviceDontPropagateList
        window: WINDOW
        eventclass: LISTofEVENTCLASS
        mode: {AddToList, DeleteFromList}

        Errors: Window, Class, Mode
```

This function modifies the list specifying the events that are not propagated to the ancestors of the specified window. You may use the modes **AddToList** or **DeleteFromList**.

GetDeviceDontPropagateList
    window: WINDOW

    Errors: Window
=>
    dont-propagate-list: LISTofEVENTCLASS

This function returns a list specifying the events that are not propagated to the ancestors of the specified window.

2.10. Sending Extension Events

One client program may send an event to another via the **XSendExtensionEvent** function.

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.

SendExtensionEvent
    device: DEVICE
    destination: WINDOW
    propagate: BOOL
    eventclass: LISTofEVENTCLASS
    event: XEVENT

Errors: Device, Value, Class, Window

2.11. Getting Motion History

GetDeviceMotionEvents
    device: DEVICE
    start, stop: TIMESTAMP or CurrentTime
=>
    nevents_return: CARD32
    mode_return: {Absolute, Relative}
    axis_count_return: CARD8
    events: LISTofDEVICETIMECOORD

where

    DEVICETIMECOORD: [data:LISTofINT32 time:TIMESTAMP]

    Errors: Device, Match

This request 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 data field of the DEVICETIMECOORD structure is a sequence of data items. Each item is of type INT32, 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, which 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 and minimum values for each axis are reported by the **ListInputDevices** 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.

2.12. Changing The Core Devices

These requests are provided to change which physical device is used as the X pointer or X keyboard. Using these requests may change the characteristics of the core devices. The new pointer device may have a different number of buttons than the old one did, 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 indicies into that array. Changing the core devices could cause such client programs to behave improperly or abnormally terminate.

These requests change the X keyboard or X pointer device and generate an **ChangeDeviceNotify** event and a **MappingNotify** event. The **ChangeDeviceNotify** event is sent only to those clients that have expressed an interest in receiving that event via the **XSelectExtensionEvent** request. 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, or has not previously been opened via **OpenDevice**.

To change the X keyboard device, use the **ChangeKeyboardDevice** request. The specified device must support input class Keys (as reported in the ListInputDevices request) or the request will fail with a **BadMatch** error. 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 CloseDevice request to close the new core device will fail with a BadDevice error.

The focus state of the new keyboard is the same as the focus state of the old X keyboard. If the new keyboard was not initialized with a **FocusRec**, one is added by the **ChangeKeyboardDevice** request. The X keyboard is assumed to have a **KbdFeedbackClassRec**. If the device was initialized without a **KbdFeedbackClassRec**, one will be added by this request. The **KbdFeedbackClassRec** will specify a null routine as the control procedure and the bell procedure.

```
ChangeKeyboardDevice
        device: DEVICE

        Errors: Device, Match
=>
        status: Success, AlreadyGrabbed, Frozen
```

To change the X pointer device, use the **ChangePointerDevice** request. The specified device must support input class Valuators (as reported in the ListInputDevices request) or the request will fail with a BadMatch error. The valuators to be used as the x- and y-axes of the pointer device must be specified. Data from other valuators on the device will be ignored.

The X pointer device does not contain a **FocusRec**. If the new pointer was initialized with a **FocusRec**, it is freed by the **ChangePointerDevice** request. The X pointer is assumed to have a **ButtonClassRec** and a **PtrFeedbackClassRec**. If the device was initialized without a **ButtonClassRec** or a **PtrFeedbackClassRec**, one will be added by this request. The **ButtonClassRec** added will have no buttons, and the **PtrFeedbackClassRec** will specify a null routine as the control procedure.

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.

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 CloseDevice request to close the new core device will fail with a BadDevice error.

ChangePointerDevice
device: DEVICE
xaxis: CARD8
yaxis: CARD8

Errors: Device, Match

=>

status: Success, AlreadyGrabbed, Frozen

2.13. 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.

2.14. Extension Active Grabs

Active grabs of extension devices are supported via the **GrabDevice** request in the same way that core devices are grabbed using the core GrabKeyboard request, except that a *Device* is passed as a function parameter. A list of events that the client wishes to receive is also passed. The **UngrabDevice** request allows a previous active grab for an extension device to be released.

To grab an extension device, use the **GrabDevice** request. The device must have previously been opened using the **OpenDevice** request.

GrabDevice
device: DEVICE
grab-window: WINDOW
owner-events: BOOL
event-list: LISTofEVENTCLASS
this-device-mode: {Synchronous, Asynchronous}
other-device-mode: {Synchronous, Asynchronous}
time:TIMESTAMP or CurrentTime
=>
status: Success, AlreadyGrabbed, Frozen, InvalidTime, NotViewable

Errors: Device, Window, Value

This request actively grabs control of the specified input device. Further input events from this device are reported only to the grabbing client. This request overrides any previous active grab by this client for this device.

The event-list parameter is a pointer to a list of event classes. These are used to indicate which events the client wishes to receive while the device is grabbed. Only event classes obtained from the grabbed device are valid.

If owner-events is **False**, input events generated from this device are reported with respect to grab-window, and are only reported if selected by being included in the 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 selected by being included in the event-list. For either value of owner-events, unreported events are discarded.

If this-device-mode is **Asynchronous**, device event processing continues normally. If the device is currently frozen by this client, then processing of device events is resumed. If this-device-mode is **Synchronous**, the state of the grabbed device (as seen by means of the protocol) appears to freeze, and no further device events are generated by the server until the grabbing client issues a releasing **AllowDeviceEvents** request 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 other-device-mode is **Asynchronous**, event processing is unaffected by activation of the grab. If other-device-mode is **Synchronous**, the state of all input devices except the grabbed one (as seen by means of the protocol) appears to freeze, and no further events are generated by the server until the grabbing client issues a releasing **AllowDeviceEvents** request or until the device grab is released. Actual events are not lost while the devices are frozen; they are simply queued for later processing.

This request generates **DeviceFocusIn** and **DeviceFocusOut** events.

This request fails and returns:

- **AlreadyGrabbed** If the device is actively grabbed by some other client.
- **NotViewable** If grab-window is not viewable.
- **InvalidTime** If the specified time is earlier than the last-grab-time for the specified device or later than the current X server time. 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.
- **Frozen** If the device is frozen by an active grab of another client.

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.

To release a grab of an extension device, use **UngrabDevice**.

UngrabDevice
    device: DEVICE
    time: TIMESTAMP or CurrentTime

    Errors: Device

This request releases the device if this client has it actively grabbed (from either **GrabDevice** or **GrabDeviceKey**) and releases any queued events. If any devices were frozen by the grab, **UngrabDevice** thaws them. The request has no effect if the specified time is earlier than the last-device-grab time or is later than the current server time.

This request generates **DeviceFocusIn** and **DeviceFocusOut** events.

An **UngrabDevice** is performed automatically if the event window for an active device grab becomes not viewable.

2.15. Passively Grabbing A Key

Passive grabs of buttons and keys on extension devices are supported via the **GrabDeviceButton** and **GrabDeviceKey** requests. These passive grabs are released via the **UngrabDeviceKey** and **UngrabDeviceButton** requests.

To passively grab a single key on an extension device, use **GrabDeviceKey**. That device must have previously been opened using the **OpenDevice** request.

GrabDeviceKey
    device: DEVICE
    keycode: KEYCODE or AnyKey
    modifiers: SETofKEYMASK or AnyModifier
    modifier-device: DEVICE or NULL
    grab-window: WINDOW
    owner-events: BOOL
    event-list: LISTofEVENTCLASS
    this-device-mode: {Synchronous, Asynchronous}
    other-device-mode: {Synchronous, Asynchronous}

    Errors: Device, Match, Access, Window, Value

This request is analogous to the core **GrabKey** request. It 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 **GrabDevice**, 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 **GrabDevice**. The active grab is terminated automatically when logical state of the device has the specified key released (independent of the logical state of the modifier keys).

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

A modifier of **AnyModifier** is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). It is not required that all modifiers specified have currently assigned keycodes. A key of **AnyKey** is equivalent to issuing the request for all possible keycodes. Otherwise, the key must be in the range specified by min-keycode and max-keycode in the **ListInputDevices** request. If it is not within that range, **GrabDeviceKey** generates a **Value** error.

NULL may be passed for the modifier_device. If the modifier_device is **NULL**, the core X keyboard is used as the modifier_device.

An **Access** error is generated if some other client has issued a **GrabDeviceKey** 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 **Access** error and no grabs are established if there is a conflicting grab for any combination.

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

To release a passive grab of a single key on an extension device, use **UngrabDeviceKey**.

UngrabDeviceKey

> device: DEVICE
> keycode: KEYCODE or AnyKey
> modifiers: SETofKEYMASK or AnyModifier
> modifier-device: DEVICE or NULL
> grab-window: WINDOW
>
> Errors: Device, Match, Window, Value, Alloc

This request is analogous to the core **UngrabKey** request. It releases the key combination on the specified window if it was grabbed by this client. A modifier of **AnyModifier** is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). A key of **AnyKey** is equivalent to issuing the request for all possible keycodes. This request has no effect on an active grab.

NULL may be passed for the modifier_device. If the modifier_device is **NULL**, the core X keyboard is used as the modifier_device.

2.16. Passively Grabbing A Button

To establish a passive grab for a single button on an extension device, use **GrabDeviceButton**.

GrabDeviceButton

> device: DEVICE
> button: BUTTON or AnyButton
> modifiers: SETofKEYMASK or AnyModifier
> modifier-device: DEVICE or NULL
> grab-window: WINDOW
> owner-events: BOOL
> event-list: LISTofEVENTCLASS
> this-device-mode: {Synchronous, Asynchronous}
> other-device-mode: {Synchronous, Asynchronous}

Errors: Device, Match, Window, Access, Value

This request is analogous to the core **GrabButton** request. It establishes an explicit passive grab for a button on an extension input device. Since 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 **OpenDevice** request.

The **GrabDeviceButton** request establishes a passive grab on a device. Consequently, in the future,

- IF the device is not grabbed and the specified button is logically pressed when the specified modifier keys logically are down (and no other buttons or modifier keys are down),
- AND the grab window contains the device,
- AND a passive grab on the same device and button/ key combination does not exist on any ancestor of the grab window,
- THEN the device is actively grabbed, as for **GrabDevice**, 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 **GrabDevice**. 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. It is not required that the specified button be assigned to a physical button.

NULL may be passed for the modifier_device. If the modifier_device is **NULL**, the core X keyboard is used as the modifier_device.

An **Access** error is generated if some other client has issued a **GrabDeviceButton** 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 **Access** error and no grabs are established if there is a conflicting grab for any combination. The request has no effect on an active grab.

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

To release a passive grab of a button on an extension device, use **UngrabDeviceButton**.

UngrabDeviceButton

device: DEVICE
button: BUTTON or AnyButton
modifiers: SETofKEYMASK or AnyModifier
modifier-device: DEVICE or NULL
grab-window: WINDOW

Errors: Device, Match, Window, Value, Alloc

This request is analogous to the core UngrabButton request. It releases the passive button/key combination on the specified window if it was grabbed by the client. A modifiers of **AnyModifier** is equivalent to issuing the request for all possible modifier combinations (including the combination of no modifiers). A button of **AnyButton** is equivalent to issuing the request for all possible buttons. This request has no effect on an active grab. The device must have previously been opened using the **OpenDevice** request otherwise a **Device** error will be generated.

NULL may be passed for the modifier_device. If the modifier_device is **NULL**, the core X keyboard is used as the modifier_device.

This request cannot be used to ungrab a button on the X pointer device. The core **UngrabButton** request should be used for that purpose.

2.17. Thawing A Device

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

```
AllowDeviceEvents
        device: DEVICE
        event-mode: {AsyncThisDevice, SyncThisDevice, AsyncOtherDevices, ReplayThisdev-
        ice, AsyncAll, or SyncAll}
        time:TIMESTAMP or CurrentTime

        Errors: Device, Value
```

The **AllowDeviceEvents** request releases some queued events if the client has caused a device to freeze. The request has no effect if the specified time is earlier than the last-grab time of the most recent active grab for the client, or if the specified time is later than the current X server time.

The following describes the processing that occurs depending on what constant you pass to the event-mode argument:

- If the specified device is frozen by the client, event processing for that device continues as usual. If the device is frozen multiple times by the client on behalf of multiple separate grabs, AsyncThisDevice thaws for all. AsyncThisDevice has no effect if the specified device is not frozen by the client, but the device need not be grabbed by the client.
- If the specified device is frozen and actively grabbed by the client, event processing for that device continues normally until the next button or key 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.
- 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 (towards 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.
- 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.
- 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 (button event for the grabbed device, key or motion event for the 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 only freeze each device once).

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

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

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

2.18. Controlling Device Focus

The current focus window for an extension input device can be determined using the **GetDeviceFocus** request. Extension devices are focused using the **SetDeviceFocus** request in the same way that the keyboard is focused using the **SetInputFocus** request, except that a device is specified as part of the request. 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 **GetDeviceFocus**.

GetDeviceFocus

    device: DEVICE

=>

    focus: WINDOW, PointerRoot, FollowKeyboard, or None
    revert-to: Parent, PointerRoot, FollowKeyboard, or None
    focus-time: TIMESTAMP

    Errors: Device, Match

This request returns the current focus state, revert-to state, and last-focus-time of an extension device.

To set the focus of an extension device, use **SetDeviceFocus**.

SetDeviceFocus

    device: DEVICE
    focus: WINDOW, PointerRoot, FollowKeyboard, or None
    revert-to: Parent, PointerRoot, FollowKeyboard, or None
    focus-time: TIMESTAMP

    Errors: Device, Window, Value, Match

This request changes the focus for an extension input device and the last-focus-change-time. The request 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, with CurrentTime replaced by the current server time.

The action taken by the server when this request 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 a window ID is assigned to the focus argument, 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 you assign **PointerRoot** to the focus argument, the focus window is dynamically taken to be the root window of whatever screen the pointer is on at each input event. In this case, the revert-to argument is ignored.
- If you assign **FollowKeyboard** to the focus argument, 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 of the request (else a **Match** error). If the focus window later becomes not viewable, the X server evaluates the revert-to argument to determine the new focus window.

- If you assign **RevertToParent** to the revert-to argument, the focus reverts to the parent (or the closest viewable ancestor), and the new revert-to value is taken to be **RevertToNone**.
- If you assign **RevertToPointerRoot**, **RevertToFollowKeyboard**, or **RevertToNone** to the revert-to argument, 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.

This request causes the X server to generate **DeviceFocusIn** and **DeviceFocusOut** events.

2.19. Controlling Device Feedback

To get the settings of feedbacks on an extension device, use **GetFeedbackControl**. This request provides functionality equivalent to the core **GetKeyboardControl** and **GetPointerControl** functions. It also provides a way to control displays associated with an input device that are capable of displaying an integer or string.

```
GetFeedbackControl
        device: DEVICE
=>
        num_feedbacks_return: CARD16
        return_value: LISTofFEEDBACKSTATE

        where

                FEEDBACKSTATE:    {KbdFeedbackState, PtrFeedbackState, IntegerFeedbackState,
                                  StringFeedbackState, BellFeedbackState, LedFeedbackState}
```

Feedbacks are reported by class. Those feedbacks that are reported for the core keyboard device are in class **KbdFeedback**, and are returned in the **KbdFeedbackState** structure. The members of that structure are as follows:

```
        CLASS Kbd:    [class: CARD8
                       length: CARD16
                       feedback id: CARD8
                       key_click_percent: CARD8
                       bell_percent: CARD8
                       bell_pitch: CARD16
                       bell_duration: CARD16
```

```
                    led_value: BITMASK
                    global_auto_repeat: {AutoRepeatModeOn, AutoRepeatMode-
                   Off}
                    auto_repeats: LISTofCARD8]
```

Those feedbacks that are equivalent to those reported for the core pointer are in feedback class **PtrFeedback** and are reported in the **PtrFeedbackState** structure. The members of that structure are:

```
CLASS Ptr:    [class: CARD8
               length: CARD16
               feedback id: CARD8
               accelNumerator: CARD16
               accelDenominator: CARD16
               threshold: CARD16]
```

Some input devices provide a means of displaying an integer. Those devices will support feedback class **IntegerFeedback**, which is reported in the **IntegerFeedbackState** structure. The members of that structure are:

```
CLASS Integer:    [class: CARD8
                   length: CARD16
                   feedback id: CARD8
                   resolution: CARD32
                   min-val: INT32
                   max-val: INT32]
```

Some input devices provide a means of displaying a string. Those devices will support feedback class **StringFeedback**, which is reported in the **StringFeedbackState** structure. The members of that structure are:

```
CLASS String:    [class: CARD8
                  length: CARD16
                  feedback id: CARD8
                  max_symbols: CARD16
                  num_keysyms_supported: CARD16
                  keysyms_supported: LISTofKEYSYM]
```

Some input devices contain a bell. Those devices will support feedback class **BellFeedback**, which is reported in the **BellFeedbackState** structure. The members of that structure are:

```
CLASS Bell:    [class: CARD8
                length: CARD16
                feedback id: CARD8
                percent: CARD8
                pitch: CARD16
                duration: CARD16]
```

The percent sets the base volume for the bell between 0 (off) and 100 (loud) inclusive, if possible. Setting to −1 restores the default. Other negative values generate a **Value** error.

The pitch sets the pitch (specified in Hz) of the bell, if possible. Setting to −1 restores the default. Other negative values generate a **Value** error.

The duration sets the duration (specified in milliseconds) of the bell, if possible. Setting to −1 restores the default. Other negative values generate a **Value** error.

A bell generator connected with the console but not directly on the device is treated as if it were part of the device. Some input devices contain LEDs. Those devices will support feedback class **Led**, which is reported in the **LedFeedbackState** structure. The members of that structure are:

```
CLASS Led:     [class: CARD8
                length: CARD16
                feedback id: CARD8
                led_mask: BITMASK
                led_value: BITMASK]
```

Each bit in led_mask indicates that the corresponding led is supported by the feedback. At most 32 LEDs per feedback are supported. No standard interpretation of LEDs is defined.

This function will fail with a **BadMatch** error if the device specified in the request does not support feedbacks.

Errors: Device, Match

To change the settings of a feedback on an extension device, use **ChangeFeedbackControl**.

```
ChangeFeedbackControl
        device: DEVICE
        feedbackid: CARD8
        value-mask: BITMASK
        value: FEEDBACKCONTROL

        Errors: Device, Match, Value
```

FEEDBACKCONTROL: {KBDFEEDBACKCONTROL, PTRFEEDBACKCONTROL, INTEGERFEEDBACKCONTROL, STRINGFEEDBACKCONTROL, BELLFEEDBACKCONTROL, LEDFEEDBACKCONTROL}

Feedback controls are grouped by class. Those feedbacks that are equivalent to those supported by the core keyboard are controlled by feedback class **KbdFeedbackClass** using the **KbdFeedbackControl** structure. The members of that structure are:

```
KBDFEEDBACKCTL:    [class: CARD8
                    length: CARD16
                    feedback id: CARD8
                    key_click_percent: INT8
                    bell_percent: INT8
                    bell_pitch: INT16
                    bell_duration: INT16
                    led_mask: INT32
                    led_value: INT32
                    key: KEYCODE
                    auto_repeat_mode: {AutoRepeatModeOn, AutoRepeatModeOff, AutoRepeatModeDefault}]
```

The key_click_percent sets the volume for key clicks between 0 (off) and 100 (loud) inclusive, if possible. Setting to −1 restores the default. Other negative values generate a **Value** error.

If both auto_repeat_mode and key are specified, then the auto_repeat_mode of that key is changed, if possible. If only auto_repeat_mode is specified, then the global auto-repeat mode for the entire keyboard is changed, if possible, without affecting the per-key settings. It is a **Match** error if a key is specified without an auto_repeat_mode.

The order in which controls are verified and altered is server-dependent. If an error is generated, a subset of the controls may have been altered.

Those feedback controls equivalent to those of the core pointer are controlled by feedback class **PtrFeedbackClass** using the **PtrFeedbackControl** structure. The members of that structure are as follows:

```
PTRFEEDBACKCTL:     [class: CARD8
                     length: CARD16
                     feedback id: CARD8
                     accelNumerator: INT16
                     accelDenominator: INT16
                     threshold: INT16]
```

The acceleration, expressed as a fraction, is a multiplier for movement. For example, specifying 3/1 means the device moves three times as fast as normal. The fraction may be rounded arbitrarily by the X server. Acceleration only takes effect if the device moves more than threshold pixels at once and only applies to the amount beyond the value in the threshold argument. Setting a value to -1 restores the default. The values of the do-accel and do-threshold arguments must be nonzero for the device values to be set. Otherwise, the parameters will be unchanged. Negative values generate a **Value** error, as does a zero value for the accel-denominator argument.

Some devices are capable of displaying an integer. This is done using feedback class **IntegerFeedbackClass** using the **IntegerFeedbackControl** structure. The members of that structure are as follows:

```
INTEGERCTL:     [class: CARD8
                 length: CARD16
                 feedback id: CARD8
                 int_to_display: INT32]
```

Some devices are capable of displaying an string. This is done using feedback class **StringFeedbackClass** using the **StringFeedbackCtl** structure. The members of that structure are as follows:

```
STRINGCTL:     [class: CARD8
                length: CARD16
                feedback id: CARD8
                syms_to_display: LISTofKEYSYMS]
```

Some devices contain a bell. This is done using feedback class **BellFeedbackClass** using the **BellFeedbackControl** structure. The members of that structure are as follows:

```
BELLCTL:     [class: CARD8
              length: CARD16
              feedback id: CARD8
              percent: INT8
```

pitch: INT16
duration: INT16]

Some devices contain leds. These can be turned on and off using the **LedFeedbackControl** structure. The members of that structure are as follows:

LEDCTL: [class: CARD8
length: CARD16
feedback id: CARD8
led_mask: BITMASK
led_value: BITMASK]

Errors: Device, Match, Value

2.20. Ringing a Bell on an Input Device

To ring a bell on an extension input device, use **DeviceBell**.

DeviceBell
device: DEVICE
feedbackclass: CARD8
feedbackid: CARD8
percent: INT8
Errors: Device, Value

This request is analogous to the core **Bell** request. 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 **Value** 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 **ChangeFeedbackControl** request.

2.21. Controlling Device Encoding

To get the keyboard mapping of an extension device that has keys, use **GetDeviceKeyMapping**.

GetDeviceKeyMapping
device: DEVICE
first-keycode: KEYCODE
count: CARD8
=>
keysyms-per-keycode: CARD8
keysyms: LISTofKEYSYM

Errors: Device, Match, Value

This request returns the symbols for the specified number of keycodes for the specified extension device, starting with the specified keycode. The first-keycode must be greater than or equal to min-keycode as returned in the connection setup (else a **Value** error), and

first-keycode + count − 1

must be less than or equal to max-keycode as returned in the connection setup (else a **Value** error). The number of elements in the keysyms list is

count * keysyms-per-keycode

and KEYSYM number N (counting from zero) for keycode K has an index (counting from zero) of

(K − first-keycode) * keysyms-per-keycode + N

in keysyms. The keysyms-per-keycode 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.

If the specified device has not first been opened by this client via **OpenDevice**, or if that device does not support input class Keys, this request will fail with a **Device** error.

To change the keyboard mapping of an extension device that has keys, use **ChangeDeviceKey-Mapping**.

ChangeDeviceKeyMapping
device: DEVICE
first-keycode: KEYCODE
keysyms-per-keycode: CARD8
keysyms: LISTofKEYSYM
num_codes: CARD8

Errors: Device, Match, Value, Alloc

This request is analogous to the core **ChangeKeyMapping** request. It defines the symbols for the specified number of keycodes for the specified extension device. If the specified device has not first been opened by this client via **OpenDevice**, or if that device does not support input class Keys, this request will fail with a **Device** error.

The number of elements in the keysyms list must be a multiple of keysyms_per_keycode. Otherwise, **ChangeDeviceKeyMapping** generates a **Length** 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 **Value** 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 **Value** error:

first_keycode + (num_codes / keysyms_per_keycode) - 1

To obtain the keycodes that are used as modifiers on an extension device that has keys, use **GetDeviceModifierMapping**.

GetDeviceModifierMapping
device: DEVICE
=>
keycodes-per-modifier: CARD8
keycodes: LISTofKEYCODE

Errors: Device, Match

This request is analogous to the core **GetModifierMapping** request. This request returns the keycodes of the keys being used as modifiers. The number of keycodes in the list is 8*keycodes-per-modifier. The keycodes are divided into eight sets, with each set containing keycodes-per-modifier elements. The sets are assigned in order to the modifiers **Shift**, **Lock**, **Control**, **Mod1**, **Mod2**, **Mod3**, **Mod4**, and **Mod5**. The keycodes-per-modifier value is chosen arbitrarily by the server; zeroes are used to fill in unused elements within each set. If only zero values are given in a set, the use of the corresponding modifier has been disabled. The order of keycodes within each set is chosen arbitrarily by the server.

To set which keycodes that are to be used as modifiers for an extension device, use **SetDeviceModifierMapping**.

SetDeviceModifierMapping

device: DEVICE
keycodes-per-modifier: CARD8
keycodes: LISTofKEYCODE

=>

status: {Success, Busy, Failed}

Errors: Device, Match, Value, Alloc

This request is analogous to the core **SetModifierMapping** request. This request specifies the keycodes (if any) of the keys to be used as modifiers. The number of keycodes in the list must be 8*keycodes-per-modifier (else a **Length** error). The keycodes are divided into eight sets, with the sets, with each set containing keycodes-per-modifier elements. The sets are assigned in order to the modifiers **Shift**, **Lock**, **Control**, **Mod1**, **Mod2**, **Mod3**, **Mod4**, and **Mod5**. Only non-zero keycode values are used within each set; zero values are ignored. All of the non-zero keycodes must be in the range specified by min-keycode and max-keycode in the **ListInputDevices** request (else a **Value** error). The order of keycodes within a set does not matter. If no non-zero values are specified in a set, the use of the corresponding modifier is disabled, and the modifier bit will always be zero. Otherwise, the modifier bit will be one whenever at least one of the keys in the corresponding set is in the down position.

A 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 keys per modifier are not supported). The status reply is **Failed** if some such restriction is violated, and none of the modifiers are changed.

If the new non-zero keycodes specified for a modifier differ from those currently defined, and any (current or new) keys for that modifier are logically in the down state, then the status reply is **Busy**, and none of the modifiers are changed.

> This request generates a DeviceMappingNotify event on a **Success** status. The **DeviceMappingNotify** event will be sent only to those clients that have expressed an interest in receiving that event via the **XSelectExtensionEvent** request.

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.

2.22. Controlling Button Mapping

These requests are analogous to the core **GetPointerMapping** and **ChangePointerMapping** requests. They allow a client to determine the current mapping of buttons on an extension device, and to change that mapping.

To get the current button mapping for an extension device, use **GetDeviceButtonMapping**.

```
GetDeviceButtonMapping
        device: DEVICE
        nmap: CARD8
=>
        map_return: LISTofCARD8

        Errors:  Device, Match
```

The **GetDeviceButtonMapping** function returns the current mapping of the buttons on the specified device. Elements of the list are indexed starting from one. The length of the list indicates the number of physical buttons. The nominal mapping is the identity mapping map[i]=i.

nmap indicates the number of elements in the **map_return** array. Only the first nmap entries will be copied by the library into the map_return array.

To set the button mapping for an extension device, use **SetDeviceButtonMapping**.

```
SetDeviceButtonMapping
        device: DEVICE
        map: LISTofCARD8
        nmap: CARD8

=>
        status: CARD8

        Errors:  Device, Match, Value
```

The **SetDeviceButtonMapping** function sets the mapping of the specified device and causes the X server to generate a **DeviceMappingNotify** event on a status of **MappingSuccess**. Elements of the list are indexed starting from one. The length of the list, specified in **nmap**, must be the same as **GetDeviceButtonMapping** would return. Otherwise, **SetDeviceButtonMapping** generates a **Value** error. A zero element disables a buttons, and elements are not restricted in value by the number of physical buttons. However, no two elements can have the same nonzero value. Otherwise, this function generates a **Value** error. If any of the buttons to be altered are in the down state, the status reply is **MappingBusy** and the mapping is not changed.

2.23. Obtaining The State Of A Device

To obtain vectors that describe the state of the keys, buttons and valuators of an extension device, use **QueryDeviceState**.

```
QueryDeviceState
        device: DEVICE
=>
        device-id: CARD8
        data: LISTofINPUTCLASS
```

where

INPUTCLASS: {VALUATOR, BUTTON, KEY}

CLASS VALUATOR: [class: CARD8
num_valuators: CARD8
mode: CARD8
#x01 device mode
(0 = Relative, 1 = Absolute)
#x02 proximity state
(0 = InProximity, 1 = OutOfProximity)
valuators: LISTofINT32]

CLASS BUTTON: [class: CARD8
num_buttons: CARD8
buttons: LISTofCARD8]

CLASS KEY: [class: CARD8
num_keys: CARD8
keys: LISTofCARD8]

Errors: Device

The **QueryDeviceState** request returns the current logical state of the buttons, keys, and valuators on the specified input device. The *buttons* and *keys* arrays, byte N (from 0) contains the bits for key or button 8N to 8N+7 with the least significant bit in the byte representing key or button 8N.

If the device has valuators, a bit in the mode field indicates whether the device is reporting Absolute or Relative data. If it is reporting Absolute data, the valuators array will contain the current value of the valuators. If it is reporting Relative data, the valuators array will contain undefined data.

If the device reports proximity information, a bit in the mode field indicates whether the device is InProximity or OutOfProximity.

3. 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.

3.1. Button, Key, and Motion Events

DeviceKeyPress
DeviceKeyRelease
DeviceButtonPress,
DeviceButtonRelease
DeviceMotionNotify

device: CARD8
root, event: WINDOW
child: Window or None
same-screen: BOOL
root-x, root-y, event-x, event-y: INT16
detail: <see below>
state: SETofKEYBUTMASK
time: TIMESTAMP

These events are generated when a key, button, or valuator logically changes state. The generation of these logical changes may lag the physical changes, if device event processing is frozen. Note that **DeviceKeyPress** and **DeviceKeyRelease** are generated for all keys, even those mapped to modifier bits. The ‘‘source’’ of the event is the window the pointer is in. The window with respect to which the event is normally reported is found by looking up the hierarchy (starting with the source window) for the first window on which any client has selected interest in the event. The actual window used for reporting can be modified by active grabs and by the focus window.The window the event is reported with respect to is called the ‘‘event’’ window.

The root is the root window of the ‘‘source’’ window, and root-x and root-y are the pointer coordinates relative to root’s origin at the time of the event. Event is the ‘‘event’’ window. If the event window is on the same screen as root, then event-x and event-y are the pointer coordinates relative to the event window’s origin. Otherwise, event-x and event-y are zero. If the source window is an inferior of the event window, then child is set to the child of the event window that is an ancestor of (or is) the source window. Otherwise, it is set to None. The state component gives the logical state of the buttons on the core X pointer and modifier keys on the core X keyboard just before the event. The detail component type varies with the event type:

Event	Component
DeviceKeyPress, DeviceKeyRelease	KEYCODE
DeviceButtonPress, DeviceButtonRelease	BUTTON
DeviceMotionNotify	{ Normal , Hint }

The granularity of motion events is not guaranteed, but a client selecting for motion events is guaranteed to get at least one event when a valuator changes. If **DeviceMotionHint** is selected, the server is free to send only one **DeviceMotionNotify** event (with detail **Hint**) to the client for the event window, until either a key or button changes state, the pointer leaves the event window, or the client issues a **QueryDeviceState** or **GetDeviceMotionEvents** request.

3.2. DeviceValuator Event

DeviceValuator

device: CARD8
device_state: SETofKEYBUTMASK
num_valuators: CARD8
first_valuator: CARD8
valuators: LISTofINT32

DeviceValuator events are generated to contain valuator information for which there is insufficient space in DeviceKey, DeviceButton, DeviceMotion, and Proximity wire events. For events of these types, a second event of type DeviceValuator follows immediately. The library combines these events into a single event that a client can receive via XNextEvent. DeviceValuator events are not selected for by clients, they only exist to contain information that will not fit into some event selected by clients.

The device_state component gives the state of the buttons and modifiers on the device generating the event.

Extension motion devices may report motion data for a variable number of axes. The valuators array contains the values of all axes reported by the device. If more than 6 axes are reported, more than one DeviceValuator event will be sent by the server, and more than one DeviceKey, DeviceButton, DeviceMotion, or Proximity event will be reported by the library. Clients should examine the corresponding fields of the event reported by the library to determine the total number of axes reported, and the first axis reported in the current event. Axes are numbered beginning with zero.

For Button, Key and Motion events on a device reporting absolute motion data the current value of the device's valuators is reported. For devices that report relative data, Button and Key events may be followed by a DeviceValuator event that contains 0s in the num_valuators field. In this case, only the device_state component will have meaning.

3.3. Device Focus Events

DeviceFocusIn
DeviceFocusOut

> device: CARD8
> time: TIMESTAMP
> event: WINDOW
> mode: { Normal, WhileGrabbed, Grab, Ungrab}
> detail: { Ancestor, Virtual, Inferior, Nonlinear, NonlinearVirtual, Pointer, PointerRoot, None}

These events are generated when the input focus changes and are reported to clients selecting **DeviceFocusChange** for the specified device and window. Events generated by **SetDeviceFocus** when the device is not grabbed have mode **Normal**. Events generated by **SetDeviceFocus** when the device is grabbed have mode **WhileGrabbed**. Events generated when a device grab actives have mode **Grab**, and events generated when a device grab deactivates have mode **Ungrab**.

All **DeviceFocusOut** events caused by a window unmap are generated after any **UnmapNotify** event, but the ordering of **DeviceFocusOut** with respect to generated **EnterNotify**, **LeaveNotify**, **VisibilityNotify** and **Expose** events is not constrained.

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

3.4. Device State Notify Event

DeviceStateNotify

> time: TIMESTAMP
> device: CARD8
> num_keys: CARD8
> num_buttons: CARD8
> num_valuators: CARD8
> classes_reported: CARD8 {SetOfDeviceMode **|** SetOfInputClass}
> > SetOfDeviceMode:
> > > #x80 ProximityState
> > > > 0 = InProxmity, 1 = OutOfProximity
> > >
> > > #x40 Device Mode
> > > > (0 = Relative, 1 = Absolute)
> >
> > SetOfInputClass:
> > > #x04 reporting valuators
> > > #x02 reporting buttons
> > > #x01 reporting keys
>
> buttons: LISTofCARD8
> keys: LISTofCARD8
> valuators: LISTofCARD32

This event reports the state of the device just as in the **QueryDeviceState** request. This event is reported to clients selecting **DeviceStateNotify** for the device and window and is generated immediately after every **EnterNotify** and **DeviceFocusIn**. If the device has no more than 32 buttons, no more than 32 keys, and no more than 3 valuators, This event can report the state of the device. If the device has more than 32 buttons, the event will be immediately followed by a DeviceButtonStateNotify event. If the device has more than 32 keys, the event will be followed by a DeviceKeyStateNotify event. If the device has more than 3 valuators, the event will be followed by one or more DeviceValuator events.

3.5. Device KeyState and ButtonState Notify Events

DeviceKeyStateNotify

device: CARD8
keys: LISTofCARD8

DeviceButtonStateNotify

device: CARD8
buttons: LISTofCARD8

These events contain information about the state of keys and buttons on a device that will not fit into the DeviceStateNotify wire event. These events are not selected by clients, rather they may immediately follow a DeviceStateNotify wire event and be combined with it into a single DeviceStateNotify client event that a client may receive via XNextEvent.

3.6. DeviceMappingNotify Event

DeviceMappingNotify

time: TIMESTAMP
device: CARD8
request: CARD8
first_keycode: CARD8
count: CARD8

This event reports a change in the mapping of keys, modifiers, or buttons on an extension device. This event is reported to clients selecting **DeviceMappingNotify** for the device and window and is generated after every client **SetDeviceButtonMapping**, **ChangeDeviceKeyMapping**, or **ChangeDeviceModifierMapping** request.

3.7. ChangeDeviceNotify Event

ChangeDeviceNotify

device: CARD8
time: TIMESTAMP
request: CARD8

This event reports a change in the physical device being used as the core X keyboard or X pointer device. **ChangeDeviceNotify** events are reported to clients selecting **ChangeDeviceNotify** for the device and window and is generated after every client **ChangeKeyboardDevice** or **ChangePointerDevice** request.

3.8. Proximity Events

ProximityIn
ProximityOut

device: CARD8
root, event: WINDOW
child: Window or None
same-screen: BOOL
root-x, root-y, event-x, event-y: INT16
state: SETofKEYBUTMASK
time: TIMESTAMP
device-state: SETofKEYBUTMASK
axis-count: CARD8
first-axis: CARD8
axis-data: LISTofINT32

These events are generated by some devices (such as graphics tablets or touchscreens) to indicate that a stylus has moved into or out of contact with a positional sensing surface.

The ‘‘source’’ of the event is the window the pointer is in. The window with respect to which the event is normally reported is found by looking up the hierarchy (starting with the source window) for the first window on which any client has selected interest in the event. The actual window used for reporting can be modified by active grabs and by the focus window.The window the event is reported with respect to is called the ‘‘event’’ window.

The root is the root window of the ‘‘source’’ window, and root-x and root-y are the pointer coordinates relative to root’s origin at the time of the event. Event is the ‘‘event’’ window. If the event window is on the same screen as root, then event-x and event-y are the pointer coordinates relative to the event window’s origin. Otherwise, event-x and event-y are zero. If the source window is an inferior of the event window, then child is set to the child of the event window that is an ancestor of (or is) the source window. Otherwise, it is set to None. The state component gives the logical state of the buttons on the core X pointer and modifier keys on the core X keyboard just before the event. The device-state component gives the state of the buttons and modifiers on the device generating the event.