The X Keyboard Extension: Protocol Specification

Protocol Version 0.29
Working Draft

April 17, 1994

Erik Fortune
International R&D
Silicon Graphics

Section 1. Overview

This extension provides a number of new capabilities and controls for text keyboards.

The core X protocol specifies the ways that the shift, mode_switch, control and lock modifiers interact to generate KeySyms and characters and allows users to specify that a key affects one or more modifiers. This behavior is simple and fairly flexible, but it has a number of limitations that make it difficult or impossible to properly support many common varieties of keyboard behavior. The limitations of core protocol support for keyboards include:

- Use of a single, uniform, four-symbol mapping for all keyboard keys makes it difficult to properly support a numeric keypad, keyboard overlays, PC-style break keys or ISO9995 standard keyboards.
- Use of a modifier to specify a second keyboard group has a number of side-effects and wreaks havoc with client grabs and X toolkit translations and limits us to a single keyboard group.
- Poorly specified locking key behavior requires X servers to look for a few “magic” KeySyms to determine which keys should lock when pressed. This leads to a variety of incompatibilities between X servers with no way for clients to detect the differences between various implementations.
- Poorly specified capitalization and control behavior requires modifications to X library source code to support new character sets or locales and can lead to incompatibilities between system-wide and X library capitalization behavior.
- The limited interactions between modifiers specified by the core protocol make many common keyboard behaviors difficult or impossible to implement. For example, there is no reliable way to indicate whether or not the shift modifier should “cancel” the lock modifier.
- The lack of any explicit descriptions for indicators, most modifiers and other aspects of the keyboard appearance requires clients that wish to clearly describe the keyboard to a user to resort to a mish-mash of prior knowledge and heuristics.

This extension makes it possible to clearly and explicitly specify most aspects of keyboard behavior on a per-key basis. It adds the notion of a numeric keyboard group to the global keyboard state and provides mechanisms to more closely track the logical and physical state of the keyboard. For keyboard control clients, this extension provides descriptions and symbolic names for many aspects of keyboard appearance and behavior.

The X Keyboard Extension also includes a number of keyboard controls designed to make keyboards more accessible to people with movement impairments.

Section 2. Description

The X Keyboard Extension essentially replaces the core protocol definition of a keyboard. This section describes the new capabilities of the extension and the effect of the extension on core protocol requests, events and errors. Each subsection first describes some aspect of the core protocol support for keyboards and then describes the new extension behavior which replaces or extends it.

Keyboard State

The core protocol description of keyboard state consists of eight modifiers. Whenever any key which affects some modifier is physically or logically depressed, that modifier is set in the keyboard state. The protocol implies that certain modifier keys lock (i.e. affect modifier state after they have been physically released) but does not explicitly discuss locking keys or their behavior. The current modifier state is reported to clients in a number of core protocol events and can be determined using the `QueryPointer` request.

The XKB extension adds the notion of a numeric keyboard group to the keyboard state. Keyboard group is used in combination with the eight core protocol modifiers to determine how a key event affects the server or clients. A keyboard may support at most eight keyboard groups.

This extension adds explicit support for locking and latching modifiers and group state. Locked modifiers and group apply to all future key events until they are explicitly changed. Latched modifiers apply only to the next key that does not change keyboard state.

The core protocol’s interpretation of keyboard modifiers does not include support for multiple groups and might vary substantially from XKB. To make old clients behave reasonably, this extension includes a mechanism to map XKB modifier and group state into a “compatibility” state which is reported only to XKB-unaware clients. See the section titled "Alternate Symbols" below for a more complete description of this mapping.

The current effective keyboard group is reported to XKB-aware clients in the reserved bits in the state field of several core protocol events. The effective keyboard group is not directly reported to non-XKB clients, but it might affect the compatibility state reported to such clients in core protocol events.

The XKB description of keyboard state includes:

- The locked modifiers and group
- The latched modifiers and group
- The base modifiers and group (for which keys are physically or logically down)
- The effective modifiers and group (the cumulative effect of the base, locked and latched modifier and group states, see the section below titled "Other Keyboard Controls" for a more complete discussion).
- The compatibility state; this is the state that is reported to XKB-unaware clients.

XKB-aware clients can query the current keyboard state at any time or request immediate notification of a change in any of these components.

XKB includes two masks which are applied to the effective modifier state reported in core protocol events. In other words, the effective modifier state reported by the `KBGetState` request might include modifiers that are not set in the state as reported in core protocol events. The section below titled "Other Keyboard Controls" describes both of these masks in more detail.

Virtual Modifiers

The core protocol specifies the interpretation of the Shift, Lock and Control modifiers; the interpretation of the other five varies, usually depending on the keysyms bound to them. To avoid "hard-wired" definitions of these five modifiers in the various structures which describe an XKB keyboard, XKB supports sixteen "virtual" modifiers.

A virtual modifier has a name and at most one real modifier associated with it. You can use a virtual modifier to describe any aspect of keyboard behavior; whenever the value of a virtual modifier changes, XKB recalculates the corresponding real modifiers for all keyboard data structures that use the changed virtual modifier.

Unless we explicitly specify "real" or "virtual" in this document, all references to modifiers or modifier masks refer to both real and virtual modifiers. For example, most of the modifier masks used in XKB data structures consist of an eight-bit real modifier mask and a sixteen-bit virtual modifier mask; we refer to such fields as modifier masks.

Keyboard Client Map

The core protocol keyboard mapping consists of a two-dimensional array of KeySyms, one row per key. A keyboard mapping may have up to 256 columns.

The core protocol specifies a set of rules for determining which of the first four columns of a keyboard mapping should be used based on the state of the Shift modifier, the Lock modifier, and the real modifier (if any) assigned to the `Mode_switch` key. Before release 6, this mapping was the same for all keys on the keyboard; R6 added some support for numeric keypad keys.

An XKB keyboard *client map* tells clients how to map key events to KeySyms for any keyboard state. A client map consists of an array with one *key symbol map* per keycode and alist of up to 255 *key types*.

A key symbol map specifies the symbols associated with the key, the type of the key and the effect of keyboard group on the key. The symbols associated with a key can be thought of as a two-dimensional array whose height is determined by the *groupWidth* of the key type and whose width is determined by the number of groups associated with the key. The columns in this array are called *groups* and the rows are called *shift levels*.

Given the current real modifier state, a key type returns an shift level and a mask of unused real modifiers. It contains a mask which specifies the modifiers that affect keys of this type and a *groupWidth* which reports the largest shift level the key type can return. The *map* reports the shift level to be used for each significant combination of modifiers.

Modifiers that are used to select a shift level are normally consumed and are not used to determine further event processing. A key type contains an optional preserve field that indicates which of the significant modifiers should *not* be consumed by the shift level selection process.

Modifiers that are set and are not consumed by the symbol selection process may be used to determine additional transformations to be applied to the selected symbol. The

X Keyboard Extension specifies the behavior of the `Lock` and `Control` modifiers, but interpretation of other modifiers is application-dependent.

Keyboard Server Map

The core protocol defines a *modifier mapping*, which consists of one list of keycodes per modifier. Each list specifies the keys that affect the corresponding modifier. When a key is logically down, any modifiers associated with the key by the modifier mapping are set. As defined by the core X protocol, keyboard state at any point in time is essentially the union of all modifiers that are affected by one of the keys that are logically pressed.

The core protocol implies that some modifier keys logically lock when pressed, but does not describe their behavior. It does not specify any rules by which a client or server can determine or control which keys should lock and does not explain how locking keys should be implemented.

An XKB keyboard *server map* describes the way the server handles events and includes mechanisms to clearly and explicitly specify for each key:

- Whether the key locks or is a member of a radio group (*key behavior*).
- The possible effects of pressing the key on the server (*key actions*).

The X server uses key behavior to determine whether to process or filter out any given key event. Key behavior can be used to simulate (or indicate) keys that physically lock or belong to a radio group. Some key behaviors are “permanent” which means they describe a ”hard-wired” (usually physical) characteristic rather than a programmable XKB behavior. The server silently ignores attempts to change permanent behaviors.

Key behaviors and key event processing are described in detail in section 3, below.

Once the server has decided to process a key event, it applies *server actions* to determine the effects of the key on the internal state of the server. An action also consists of an operator and some optional data. XKB supports actions which:

- change base, latched or locked modifiers or group
- move the core pointer or simulate core pointer button events
- change some aspects of keyboard behavior
- terminate or suspend the server
- send a message to XKB-aware clients

Each key has an optional array of actions. If defined, it parallels the array of symbols associated with the key (i.e. it has one action per symbol associated with the key). For `KeyPress` events, the server looks up the action to be applied from this list using the key symbol mapping associated with the event key.

Server actions have essentially two halves; the effects on the server when the key is pressed and the effects when the key is released. The action applied for a `KeyPress` event determines the further actions, if any, that are applied to the corresponding `KeyRelease` event or to events that occur while the key is held down.

The *virtual modifiers* field of the server map reports the real modifiers which correspond to each of the sixteen virtual modifiers. Legal values for an entry in the virtual modifiers field include the index of any real modifier (range 0..7) or the constant `NoModifier`.

Many characteristics of a key can be determined automatically by applying symbol interpretations (described in more detail in the section titled "Compatibility Map" below). The server map includes an *explicit components* map which can force the server to retain assigned values for any or all of these components. The explicit components map has one entry per keyboard key; each entry has the following boolean fields:

explicitKeyType	If true, disables automatic determination of key type.
explicitInterpretation	If true, symbol interpretations are not compared against or applied to this key; effectively disables virtual modifier calculation and automatic determination of key actions, autorepeat and behavior.
explicitAutorepeat	If true, disables automatic determination of autorepeat for this key.
explicitBehavior	If true, disables automatic determination of behavior for this key.

Other Keyboard Controls

The core protocol provides a limited ability to control keyboard bells, autorepeat behavior, and LEDs. XKB extends this support and provides new controls over other aspects of key behavior.

The core protocol provides requests to control the pitch, volume and duration of the keyboard bell and a request to explicitly sound the bell. The *audible bell* control allows clients to disable core protocol and input extension bell requests. Extension requests also allow clients to attach an optional window and symbolic name to a bell request or to receive an event when the keyboard bell is rung.

The core protocol provides a request which can explicitly light or extinguish up to thirty-two keyboard LEDs but it does not interpret them in any way. XKB provides an *indicator map* which specifies the effect of keyboard controls or of the base, latched or locked modifier and keyboard group state on the keyboard indicators. The indicator map *prevent explicit* field contols the effect of core protocol LED requests; if *prevent explicit* is set for some indicator, its state can not be changed using the core protocol; attempts to do so are ignored.

The core protocol allows control over whether or not the entire keyboard or individual keys should autorepeat when held down. XKB adds the ability to control the rate at which repeating keys generate events. The *autorepeat delay* specifies the delay between the initial press of an autorepeating key and the first generated repeat event in milliseconds. The *autorepeat interval* specifies the delay between all subsequent generated repeat events in milliseconds.

If the *auto-autorepeat* control is set, XKB determines the autorepeat behavior of each key automatically using the symbol compatibility map (see "Effects of Core Protocol Requests on XKB" below). To control automatic autorepeat determination on a per-key basis, use explicit components (see "Keyboard Server Map" above).

The *keyboard group info* control specifies the current number of keyboard groups and whether out-of-range groups are wrapped or truncated back into range. The number of keyboard groups is automatically updated to reflect changes in the key symbol map.

Keyboard keys normally generate `KeyPress` and `KeyRelease` events immediately when a key changes state, though the server might ignore some of these events or synthesize others based on the behavior and action of the event in question. In addition to per-key behavior, XKB includes some global controls over the default key behavior.

These *global key behaviors* were originally part of a separate extension called AccessX. AccessX was designed to make keyboards more accessible to people with movement impairments and has been merged with the more general X Keyboard Extension to minimize the number of extensions in the world. See "AccessX: Keyboard Enhancements to the X Server for People with Movement Impairments" by Will Walker and Mark Novak for a good rationale for and discussion of these features.

- The *slow keys* control requires the user to hold a key for a specified period of time before the key is accepted. The *slow keys delay* is specified in milliseconds.
- The *bounce keys* control debounces the keyboard by ignoring key press events which occur within a given time period after a release of the same key. The *debounce delay* is specified in milliseconds.
- The *sticky keys* control changes the set modifiers and set groups actions to make all keyboard modifiers keys latch when pressed. XKB already provides the ability to make individual modifiers keys latch, but directly modifying the keyboard layout to make all modifier keys latch is both expensive and irreversible, so XKB supports a shortcut to provide this necessary capability.
- The *mouse keys* control enables any key actions that control the core pointer device. When mouse keys are not enabled, such key behaviors just generate key press and release events normally. The *mouse key default button* specifies the default button to be used by actions that generate core pointer button events.
- The *mouse keys accceleration* control enables repeated events when a key which moves the core pointer is held down. The *mouse keys delay*, *interval*, *steps to maximum acceleration*, *maximum acceleration* and *curve* values control the rate at which the pointer accelerates when the user holds a mouse key. All times are specified in milliseconds.
- The `AccessXTimeout` control automatically disables some keyboard controls if the keyboard and pointer remain idle for too long. The *accessX timeout mask* specifies the controls to be disabled automatically and the *accessX timeout* specifies the length of time in seconds the keyboard must be idle in order to automatically disable them.
- The `AccessXKeys` control enables or disables special, standard key sequences which activate many of the AccessX features.
- The `AccessXFeedback` control enables or disables special, standard sequences of keyboard bells to indicate activation or deactivation of the AccessX features or to indicate acceptance of a key when slow keys are active.

All of these controls may be explicitly enabled or disabled using the `KBSetControls` request or the set controls key action. If `AccessXKeys` is enabled many of these controls can also be turned on or off from the keyboard by entering special standard key sequences.

The core protocol does not provide any way to exclude certain modifiers from grab calculations, so locking modifiers often have unanticipated and unfortunate side-effects. XKB provides two masks which can help avoid some of these problems.

Modifiers specified in the *server internal modifiers* mask are not reported in any core protocol events, are not used to determine grabs and are not used to calculate compatibility state for XKB-unaware clients. Server internal modifiers normally affect only the action applied when a key is pressed.

The locked state of the modifiers specified in the *ignore locks* mask is not reported in most core protocol events, and is not used to activate grabs. The only core protocol events which include the locked state of the modifiers in the ignore locks mask are `KeyPress` and `KeyRelease` events that do not activate a passive grab. The compatibility state reported to XKB-unaware clients for any given core protocol event is computed from the modifier state that XKB-aware clients would see for that same event; e.g. the compatibility state reported to XKB-unaware clients in non-grab `KeyPress` and `KeyRelease` takes the locked state of all modifiers into account, but the compatibility state reported in all other core protocol events and requests might not.

Compatibility Map

In addition to providing a number of new requests, this extension replaces or extends existing core protocol requests and events. Some aspects of the this extension, such as the ability to lock any key or modifier, are visible even to clients that are unaware of the XKB extension. Other capabilities, such as control of keysym selection on a per-key basis, are available only to XKB-aware clients.

Though they do not have access to some advanced extension capabilities, the XKB extension includes compatibility mechanisms to ensure that non-XKB clients behave as expected and operate at least as well with an XKB-capable server as they do without.

There are two significant areas in which XKB state and mapping differences might be visible to XKB-unaware clients:

- The core protocol uses a modifier to choose between two keyboard groups, while this extension provides explicit support for multiple groups.
- The order of the symbols associated with any given key by XKB might not match the ordering demanded by the core protocol.

To minimize problems that might result from these differences, this extension includes ways to specify the correspondence between core protocol and XKB modifiers and symbols.

A keyboard description includes one *modifier compatibility map* per real or virtual modifier (twenty-four in all). Each such map specifies one or more groups and real modifiers that map to the corresponding compatibility modifier. Applying each of the modifier compatibility maps in turn to some modifier state and group yields a new KEYMASK suitable for XKB-unaware clients. If the group or any of the real modifiers specified by one of the modifier compatibility maps are set in the source modifier state, the modifier which corresponds to the compatibility map is set in the resulting compatibility state.

Whenever some client changes the keyboard mapping using core protocol requests, the X server generates a corresponding XKB-specific description of the keyboard map. First, It chooses key types from a canonical list of key types by examining the symbols associated with each changed key and computes the number of groups for each key using the key type and the number of symbols assigned to the key by the core protocol. Next, it examines the symbols and modifier mapping for each changed key and assigns server actions where appropriate. Finally, it updates other XKB compatibility structures that should change to reflect the changes requested by the XKB-unaware client.

Use explicit components (see "Keyboard Server Map") to disable one or more of these steps for specific keyboard keys.

Most of the process of mapping a core keyboard description into an XKB description is designed to preserve compatible behavior for pre-XKB clients and cannot be redefined by the user. Clients or users that want behavior that cannot be described using this mapping should use XKB functions directly.

Keyboard Description

The core protocol does not provide any information to clients other than that actually used to interpret events. This makes it difficult to write a client which presents the keyboard to a user in an easy-to-understand way. Such applications have to examine the vendor string and keycodes to determine the type of keyboard connected to the server and have to examine keysyms and modifier mappings to determine the effects of most modifiers (the `Shift`, `Lock` and `Control` modifiers are defined by the core protocol but no semantics are implied for any other modifiers).

This extension provides such applications with symbolic names for most components of the keyboard extension and a description of the physical layout of the keyboard.

The *keycodes* name describes the range and meaning of the keycodes returned by the keyboard in question; the *geometry* name describes the physical location, size and shape of the various keys on the keyboard. As an example to distinguish between these two names, consider function keys on PC-compatible keyboards. Function keys are sometimes above the main keyboard and sometimes to the left of the main keyboard, but the same keycode is used for the key that is logically `F1` regardless of physical position. Thus, all PC-compatible keyboards have the same keycodes name but might different geometry names.

Note Keyboard type is intended to be a very general description of the keycodes returned by a keyboard; A single keyboard type might cover keyboards with differing numbers of keys provided that the keys that all keys have the same semantics when present. For example, 101 and 102 key PC keyboards might use the same name. Applications can use the keyboard keycode range or geometry to determine which subset of the named keyboard type is in use.

The *symbols* name and list of *character sets* identify and describe the symbols bound to the keys. The symbols name is a human or application-readable description of the intended locale or usage of the keyboard with these symbols. The list of character sets identifies the all of the character sets from which one or more symbols are defined. For example, a keyboard with both Cyrillic and Latin-1 symbols might have a symbols name of “ru” (a standard code for the Russian language/Russia locale) and list the character sets “iso8859-1” and “iso8859-5.”

The *semantics* name is a symbolic description of the overall behavior of the keyboard. For example, the semantics name might indicate that a keyboard supports shift-cancels-caps behavior, uses arrow keys to control the core pointer or functions compatibly with some other system (such as a PC). The semantics name provides additional information for the user or for keyboard layout programs and is not otherwise interpreted by the X Keyboard Extension.

The *physical symbols* and *physical geometry* names are similar to the normal symbols and geometry names but are intended to describe the actual physical appearance of the keyboard connected to the X server (i.e. before any remapping is applied). Layout programs can use these names to determine and restore "factory defaults" for the keyboard. X servers are allowed (but not required) to ignore attempts to set these names if they can determine that the actual keyboard has not changed or that the new names are incorrect. Servers are also allowed to change these names automatically if the initial keyboard is detectably replaced by a new, different keyboard.

XKB provides symbolic names for each of the eight keyboard modifiers and thirty-two keyboard indicators.

XKB allows keyboard layout designers or editors to assign names to key types and to each of the levels in a key type. For example, the second position on an alphabetic key might be called the "Shift" level while the second position on a numeric keypad key might be called the "Numeric" level.

Each key has a four character symbolic name which keyboard layout or description applications can use to identify similar keys on physically different keyboards. For example, some workstation vendors use keyboards that look similar to PC keyboards but which return radically different keycodes. Use of symbolic names allows us to share one symbol map for keyboards that report either set of keycodes.

XKB supports names for up to 255 radio groups on a given keyboard. These names are provided as a convenience; the server does not verify that named radio groups actually have any members or that non-empty groups actually have names.

Alternate Symbols

This extension provides a limited ability to choose keyboard layout on a per-application basis, which can be extremely useful in multilingual environments. Applications can request *alternate symbol maps*, which override the key symbol maps for some or all of the keys on the keyboard.

An alternate symbol set only changes the way a client interprets key events; neither key action nor behavior are affected.

XKB Events and Detailed Event Masks

The XKB extension provides events which report changes in any aspect of the XKB keyboard state, mappings or controls and which report additional information about some key events. Most events can be caused by changes in any of a number of components or for any of a number of reasons, so each type of event has a *details* mask which allows clients to request a subset of the possible causes for a particular event.

For example, the XKB keyboard mapping contains many components, but most clients are only concerned about changes to the components that directly affect the way they interpret key events: key types and key symbol maps. Such clients can select `XkbMapNotify` events with `XkbKeyTypesMask|XkbKeySymsMask` details and XKB will only report changes in those two map components to that client.

The description of the `KBSelectEvents` request contains a complete list of XKB event types and details for each type.

Section 3. Request and Key Event Processing

Key Event Processing in the Client

The state field of key events reported to XKB-aware clients contains the effective keyboard modifier state and group. The effective modifier state is reported in the low eight bits of the state field for compatibility with the core protocol. The effective group is reported in bits 13-15 of the state field, which are reserved by the core protocol.

The key symbol map associated with the key reported in a `KeyPress` or `KeyRelease` event is used to determine the symbol generated by the event. Determine the group used to look up a symbol as follows:

```
if ( effectiveGroup < nGroups)
        group = effectiveGroup
else if ( groupsWrap )
        group = effectiveGroup % nGroups
else    group = nGroups - 1
```

where:

```
% = integer modulus
effectiveGroup is taken from the keyboard state as reported in the event
nGroups and groupsWrap are from the key symbol map of the event key
```

The offset of the desired symbol within the group is determined by the key type and the effective modifier state, so the index of the symbol associated with an event is:

```
index = ( group × type→groupWidth ) + type→map( effectiveMods & type→mask )
```

where:

```
& = bitwise AND
```

The state reported in the key event may specify additional transformations to be applied to the symbol before it is used. The *preserve* field of the key type specifies which of the modifiers that were used to compute the symbol offset should be taken into account for additional transformations:

```
consume = type→mask & ~type→preserve( effectiveMods & type→mask );
transform = ( effectiveMods & ~consume )
```

where:

```
~ = bitwise invert
```

This extension specifies the transformations to be applied when the `Control` or `Lock` modifiers are set in the remaining modifiers:

Modifier	***Transformation***
`Control`	Report the control character associated with the symbol. This extension defines the control characters associated with the ASCII alphabetic characters (both upper and lower case) and the KeySyms `atsign`, `left_bracket`, `backslash`, `right_bracket`, `asciicircum` and `underbar` (see Appendix B). Applications are free to associate control characters with any symbols that are not specified by this extension.
`Lock`	Capitalize the symbol either according to capitalization rules appropriate to the application locale or using the capitalization rules defined by this extension (see Appendix B).

Interpretation of other modifiers is application dependent.

Key Event Processing in the Server

When the X Keyboard Extension receives a key event from a keyboard, it first checks the global key behavior to decide whether to process the event immediately or at all. The global key behaviors which can affect the event, in descending order of priority, are:

- If a key is pressed while the bounce keys control is enabled, the extension generates an event only if the key is *active*. When a key is released, the server deactivates the key and starts a *bounce keys timer* with an interval specified by the debounce delay.
  If the bounce keys timer expires or if some other key is pressed before the timer expires, the server reactivates the corresponding key and deactivates the timer. Neither expiration nor deactivation of a bounce keys timer causes an event.
- If the slow keys control is enabled, the extension sets a *slow keys timer* with an interval specified by the slow keys delay, but does not process the key event immediately. The corresponding release deactivates the timer and causes a `KBSlowKeyNotify` event.
  If the slow keys timer expires, the server generates a key press for the corresponding key, sends a `KBSlowKeyNotify` and deactivates the timer.
- If the repeat keys control for the keyboard is enabled, the extension processes key press events normally, but it also initiates an *autorepeat timer* with an interval specified by the autorepeat delay if per-key autorepeat is enabled for the event key. The corresponding key release deactivates the timer.
  If the autorepeat timer expires, the server generates a key release and a key press for the corresponding key and reschedules the timer according to the autorepeat interval.

Timer-generated events report the sequence number, time and position at which the timer expired. The repeat keys, slow keys and bounce keys controls are not applied to timer initiated events, but per-key behavior and key actions are handled normally.

Once a timer expires or the server decides to process an event based on the global key controls, it checks the per-key behavior of the corresponding key. XKB currently defines five key behaviors:

Behavior	***Effect***
`KB_Default`	Press and release events are processed normally.
`KB_Lock` permanent: BOOL	If *permanent* is `True`, this behavior indicates a key that locks physically or in some software layer which is not under the control of XKB. Otherwise, XKB simulates a locking key as follows: If a key is logically up (i.e. the corresponding bit of the core key map is cleared) when it is pressed, the key press is processed normally and the corresponding release is ignored. If the key is logically down when pressed, the key press is ignored but the corresponding release is processed normally.

Behavior	*Effect*
`KB_RadioGroup` permanent: BOOL index: CARD8	If *permanent* is `True`, this behavior indicates a key that behaves as a member of a radio group physically or in a software layer that is not under the control of XKB. Otherwise, XKB simulates radio group behavior as follows: If another member of the radio group specified by *index* is logically down when a key is pressed, the server synthesizes a key release for the member that is logically down and then processes the new key press event normally. If the key itself is logically down when pressed, the key press event is ignored, but the release is processed normally. All other key release events are ignored.

Next, the server determines the server action associated with any events allowed or generated by the per-key behavior. For key press events, the server chooses an action from the list of actions associated with the key. The action associated with key release events is determined solely by the action initiated by the corresponding key press.

If the list of actions associated with any key is length zero, the server applies `SA_NoAction` for that key regardless of the keyboard modifier or group state. Otherwise, the server uses the action at the same offset as the symbol associated with the event. See "Key Event Processing in the Client" for an detailed explanation of choosing a symbol for an event.

Most actions directly change the state of the keyboard or server; some actions also modify other actions that occur simultaneously with them. Two actions occur simultaneously if the keys which invoke the actions are both logically down at the same time, regardless of the order in which they are pressed.

Some actions take an argument of type XKBMASK, which consists of the *useModMapMods* flag, a core protocol KEYMASK and a 16-bit VMODMASK which specifies one or more virtual modifiers. If *useModMapMods* is True, the XKBMASK affects any modifiers associated with the key by the core protocol modifier mask, otherwise it affects the modifiers specified by the KEYMASK and any real modifiers bound to the virtual modifiers specified by the VMODMASK. For example, if virtual modifier 1 is bound to Mod3, an XKBMASK which specifies the real modifier “Shift” and virtual modifier 1 affects both Shift and Mod3.

Version 0.29 of the X Keyboard Extension supports the following actions:

Action	*Effect*
`SA_NoAction`	• No direct effect, though `SA_NoAction` events may change the effect of other server actions (see below).
`SA_SetMods` clearLocks: BOOL mods: XKBMASK	• For key press events, sets the base modifiers specified in *mods*. • Key release clears the base modifiers in *mods*, provided that no other key which affects the same modifiers is logically down. • If no keys were operated simultaneously with the modifier shift key and *clearLocks* is set, key release also clears the locked state of any modifiers specified in *mods*.

Action	*Effect*
`SA_LatchMods` clearLocks: BOOL latchToLock: BOOL mods: XKBMASK	• Key press and key release events have the same effect as for `SA_SetMods`; if no keys were operated simultaneously with the latching modifier key, key release events have the following additional effects: • If *clearLocks* is set and takes effect, any modifiers that are unlocked have no further effect. • If *latchToLock* is set, the key release locks and unlatches any remaining *mods* that are already latched. Any such modifiers have no further effect. • Finally, key release latches any *mods* that were not used by *clearLocks* or *latchToLock*.
`SA_LockMods` mods: XKBMASK	• For key press events, sets the base and locked state of any modifiers specified in *mods*. • For key release events, clears the base modifiers specified in *mods*, provided that no other key which affects the same modifiers is down. If the modifiers specified in *mods* were locked before the corresponding key press occurred, key release unlocks them.
`SA_SetGroup` absolute: BOOL clearLocks: BOOL group: INT8	• If *absolute* is set, key press events change the base keyboard group to *group*; otherwise, they add *group* to the base keyboard group. In either case, the result is wrapped or truncated back into range depending on the value of the groupsWrap flag for the keyboard. • If an `SA_ISOLock` key is pressed while this key is held down, key release has no effect, otherwise it cancels the effects of the corresponding press. • If no keys were operated simultaneously with the group shift key *clearLocks* is set, release also sets the locked keyboard group to `0`.
`SA_LatchGroup` absolute: BOOL clearLocks: BOOL latchToLock: BOOL group: INT8	• Key press and release events have the same effect as an `SA_SetGroup` action; if no keys were operated simultaneously with the latching group key and *clearLocks* was not set or had no effect, key release has the following additional effects: • If *latchToLock* is set and the latched keyboard group is non-zero, the key release adds the delta applied by the corresponding key press to the locked keyboard group and subtracts it from the latched keyboard group. • Otherwise, key release adds the key press delta to the latched keyboard group.
`SA_LockGroup` absolute: BOOL group: INT8	• If *absolute* is set, key press sets the locked keyboard group to *group*. Otherwise, key press adds *group* to the locked keyboard group. • Key release has no effect.

Action	*Effect*
SA_AccelPtr dx, dy: INT16	• If MouseKeys are disabled, behaves like SA_NoAction. • Otherwise, key press generates a core pointer MotionNotify event instead of the usual KeyPress. The coordinates of the generated event are determined by adding *dx* and *dy* to the current core pointer position. • If the MouseKeysAccel control is enabled, key press also initiates the mouse keys timer for this key; every time this timer expires, the cursor moves again. The distance the cursor moves in these subsequent events is determined by the mouse keys acceleration. • Key release disables the mouse keys timer if set but has no other effect and is ignored (does not generate an event of any type).
SA_MovePtr dx, dy: INT16	• If MouseKeys are not enabled, this action behaves like SA_NoAction. • Otherwise, key press and release behave like the SA_MovePtr action but do not affect the mouse keys timer.
SA_PtrBtn button: CARD8	• If MouseKeys are not enabled, this action behaves like SA_NoAction. • If the mouse button specified for this action is logically down, the key press and corresponding release are ignored and have no effect. • Otherwise, key press generates a core pointer ButtonPress event instead of the usual KeyPress. If *button* has the value SA_UseDfltBtn, the event is generated for the current default core button. Otherwise, the event is generated for the button specified by *button*. • Key release generates a core pointer ButtonRelease which matches the event generated by the corresponding key press. Key release does not generate a KeyRelease event.
SA_ClickPtrBtn count: CARD8 button: CARD8	• Behaves like SA_NoAction if MouseKeys are disabled. • If the mouse button specified for this action is logically down, the key press and corresponding release are ignored and have no effect. • Otherwise, key press generates count pairs of ButtonPress and ButtonRelease events instead of the usual KeyPress event. All events are for the button specified by *button* as interpreted by the SA_ClickPtrBtn action. • Key release has no effect and is ignored.
SA_LockPtrBtn button: CARD8	• If MouseKeys are disabled, behaves like SA_NoAction. • Otherwise, if the button specified by *button* is not locked, key press causes a ButtonPress instead of a KeyPress and locks the button. If the button is already locked, key press is ignored and has no effect. • If the corresponding key press was ignored, key release generates a ButtonRelease event instead of a KeyRelease event and unlocks the specified button. If the corresponding key press locked a button, key release is ignored and has no effect.

Action	*Effect*
`SA_SetPtrDflt` affect: CARD8 flags: CARD8 value: INT8	• If `MouseKeys` are disabled, behaves like `SA_NoAction`. • Otherwise, both key press and key release are ignored, but key press changes the pointer value specified by *which* to *value*. • If *affect* is SA_AffectDfltBtn, this action changes the current default pointer button. If SA_DfltBtnAbsolute is set in *flags*, *value* specifies the default pointer button used by the various pointer actions, otherwise *value* specifies an amount to be added to the default button. • This action can cause a `KBControlsNotify` event.
`SA_SetControls` controls: CARD32	• Key press enables any controls that are specified in *controls* and not already enabled at the time of the key press. Key release disables any controls that were enabled by the corresponding key press. This action can cause `KBControlsNotify` events.
`SA_LockControls` controls: CARD32	• Key press locks and enables any controls that are specified in *controls* and not already locked at the time of the key press. Key release unlocks and disables any controls that are specified in *controls* and were not enabled at the time of the corresponding key press. This action can cause a `KBControlsNotify` event.
`SA_ISOLock` dfltIsGroup: BOOL clearLocks: BOOL latchToLock: BOOL noAffectMods: BOOL noAffectGroup:BOOL noAffectPtr: BOOL noAffectCtrls: BOOL mods: XKBMASK absolute: BOOL group: INT8	• If *dfltIsGroup* is `True`, key press sets the base group specified by *absolute* and *group*. Otherwise, key press sets the base modifiers specified in *mods*. • Key release clears the base modifiers or group that were set by the key press; it may have additional effects if no other appropriate actions occur simultaneously with the `SA_ISOLock` operation. • If *noAffectMods* is `False`, any `SA_SetMods` or `SA_LatchMods` actions that occur simultaneously with the `ISOLock` action are treated as `SA_LockMods` instead. • If *noAffectGroup* is False, any `SA_SetGroup` or `SA_Latch-Group` actions that occur simultaneously with this action are treated as `SA_LockGroup` instead. • If *noAffectPtr* is False, `SA_PtrBtn` actions that occur simultaneously with the `SA_ISOLock` action are treated as `SA_LockP-trBtn` actions instead. • If *noAffectCtrls* is set, any `SA_SetControls` actions that occur simulatneously with the `SA_ISOLock` action are treated as `SA_LockControls` actions instead. • If no other actions were transformed by the `SA_ISOLock` action, key release locks the group or modifiers specified by the action arguments.
`SA_TerminateServer`	• Key press terminates the server. Key release is ignored. • This action is optional; servers are free to ignore it. If ignored, it behaves like `SA_NoAction`.
`SA_SwitchScreen` switchApps: BOOL absolute: BOOL num: INT8	• If the server supports this action and multiple screens or displays (either virtual or real), this actions changes to the active screen indicated by *num*. If *switchApps* is `False`, the server should switch to another screen on the same display, otherwise it should switch to another application which shares the physical display. • If *absolute* is `True`, *num* specifies an absolute screen index to be used; otherwise, *num* specifies an offset from the current screen. • This action is optional; servers are free to ignore it if they do not support virtual screens. If ignored, it behaves like `SA_NoAction`, otherwise neither press nor release cause an event.

Action	*Effect*
SA_ActionMessage msgPress: BOOL msgRelese: BOOL genKeyEvent: BOOL data: LISTofCARD8	• Generates KBActionMessage events in place of or in addition to core events. • If *msgPress* is True, key press causes a KBActionMessage event; if *msgRelease* is True, key release causes an event. Any combination of *msgPress* and *msgRelease* is valid. • If *genKeyEvent* is True, the server also generates the corresponding core protocol key press and release events, otherwise it generates only the KBActionMessage events.

If StickyKeys are enabled, all SA_SetMods and SA_SetGroup actions act like SA_LatchMods and SA_LatchGroup respectively. Either event behaves as if both the *clearLocks* and *latchToLock* arguments are True.

Core events generated by server actions contain the keyboard state that was in effect at the time the key event occurred; the reported state does not reflect any changes in state that occur as a result of the actions bound to the key event that caused them.

Events sent to clients that have not issued a KB_UseExtension request contain a compatibility state in place of the actual XKB keyboard state. See "Effects of XKB on Core Protocol Requests and Events" below for a description of compatibility mapping.

If an action changes any component of the keyboard state, the server sends KBStateNotify events to any clients that have requested them.

Effects of Core Protocol Requests on XKB

In response to a core protocol ChangeKeyboardMapping request, the X server assigns a key type to each changed key from a canonical list of key types. The first three key types in any keyboard map are reserved for these standard key types:

Key Type Name	*Standard Definition*
ONE_LEVEL	Describes keys that have exactly one symbol per group. Most special or function keys (such as Return) are ONE_LEVEL keys. Any combination of modifiers yields level 0. Index 0 in any key symbol map specifies key type ONE_LEVEL.
TWO_LEVEL	Describes non-keypad keys that have exactly two symbols per group. Most alphanumeric keys are TWO_LEVEL keys. Yields column 1 if the Shift modifier is set, column 0 otherwise. Index 1 in any key symbol map specifies key type TWO_LEVEL.
KEYPAD	Describes numeric keypad keys with two symbols per group. Yields column 1 if either the Shift modifier or the *default num lock modifier* alone are set. Yields column 0 if neither or both modifiers are set. Index 2 in any key symbol map specifies key type KEYPAD.

Users or applications may redefine these key types to get different default behavior (to make shift cancel caps lock, for example) but they must always have the specified number of symbols per group.

The server does not change the assigned type of any keys for which *explicitKeyType* is set in the explicit components map. If *explicitKeyType* is not set, the server chooses a key type as follows:

Before choosing a type for a key, the X server expands any alphanumeric symbol definitions according to the core protocol rules: If the second symbol of either group is NoSymbol and the first symbol of that group is an alphabetic KEYSYM for which both lowercase and uppercase forms are defined, the X server treats the key as if the first element of the group were the lowercase form of the symbol and the second element were the uppercase form of the symbol.

If the new list of keysyms for the key has one member or the form "K1 NoSymbol K2 NoSymbol" (after alphanumeric expansion and ignoring trailing NoSymbol entries) the server assigns key type `ONE_LEVEL`. If both symbols in the first group are numeric keypad keysyms, the server assigns key type `KEYPAD`. Otherwise, it assigns key type `TWO_LEVEL`.

After choosing a key type, the server determines the number of groups associated with the key and the XKB keysym list which corresponds to the standard core protocol interpretation of the first two keyboard groups for the chosen key type.

This extension does not define the interpretation of any keysyms in a core keyboard mapping beyond the first four symbols.

Unless *explicitInterpretation* is set in the explicit components map of a changed key, the X server determines autorepeat, updates virtual modifiers and assigns actions and behaviors to the key as specified by the keyboard's symbol compatibility map. The symbol compatibility map is a list of symbol interpretation structures.

Each symbol interpretation has four fields which are compared to each keysym and the modifier mapping on any keys that have changed. The *symbol* field is a KEYSYM which matches if it has the value `NoSymbol` or is identical to the symbol in question. The modifiers associated with the symbol are compared to the modifiers specified in the *mods* field as indicated by *match*. The *match* field can specify any of the comparisions: `NoneOf`, `AnyOfOrNone`, `AnyOf`, `AllOf` or `Exactly`. The modifiers assigned to the key by the core protocol modifier mapping are associated with the symbol unless *levelOneOnly* is `True` and the symbol in question is not the first symbol bound to a key, in which case an empty modifier mask is used instead.

The server compares each keysym and the modifier mapping for all changed keys to each symbol interpretation in turn. If all three fields match, the server applies the interspretation to the key in question. A symbol interpretation applies the following fields:

Field	***Type***	***Effect***
action	`XkbAction`	Assigned to the key and symbol position in question.
autorepeat	`BOOL`	If the auto-autorepeat control is enabled, *explicitAutorepeat* is `False` in the explicit components map for this key and the symbol in question is the first symbol associated with the key, the autorepeat behavior of the key is changed to this value.

Field	*Type*	*Effect*
virtualMod	`0..15 OR NoModifier`	If this field specifies a valid virtual modifier, the highest real modifier associated with this key by the core modifier mapping is bound to the specified virtual modifier. Any XKB structures defined in terms of the specified virtual modifier are updated to reflect the new binding.
lockingKey	`BOOL`	If `True`, and if *explicitBehavior* is `False`, the behavior of the key in question is changed to `KB_Lock`.

The server considers all symbol interpretations which specify an explicit keysym before considering any that do not. The server uses the first interpretation which matches the given combination of keysym and modifier mapping; other matching interpretations are ignored. If no elements match, the server uses `SA_NoAction` and `KB_Default`.

If all of the actions computed for a key are `SA_NoAction`, the server assigns an empty list of actions to the key.

If the core protocol modifier mapping is changed, the server regenerates actions for all keyboard keys by applying the symbol compatibility map as described above. As an option, actions can also be recomputed when keyboard symbols are updated using XKB requests.

Explicitly changing the autorepeat mode of any keyboard key using the core protocol `ChangeKeyboardControl` request sets *explicitAutorepeat* for that key to prevent automatic calculation of autorepeat behavior for that key in the future.

If the audible bell is disabled, core protocol `Bell` and input extension `DeviceBell` requests cause `KBBellNotify` events but they do not generate a sound. All non-XKB bell requests generate events with name and window `None`.

If the *noExplicit* field of some indicator map is `True`, attempts to change the state of that indicator using the core protocol or input extension requests are silently ignored.

Effects of XKB on Core Protocol Requests and Events

After applying server actions which modify the base, latched or locked modifier or group state of the keyboard, the X server recomputes the effective group and state. Several variations of the effective state are reported to XKB-aware clients depending on context:

- The effective modifier state is the union of the base, latched and locked modifier states. It is reported in `KBStateNotify` events and in response to `KBGetState` requests.
- The symbol lookup state is the effective modifier state minus the server internal modifiers. It is reported to XKB-aware clients in the state field of core `KeyPress` and `KeyRelease` events that do not activate passive grabs.
- The effective grab state is the symbol lookup state minus the intersection of the locked modifier state and the ignore locks modifiers. It is reported to XKB-aware clients in the state field of all core protocol events that report keyboard state, except `KeyPress` and `KeyRelease` events that do not activate passive grabs. The server uses the effective grab state to determine whether or not to activate passive grabs requested by an XKB-aware client.

- The effective group is the sum of the base, latched and locked keyboard groups. An out of range effective group is wrapped or truncated into range according to the setting of *groupsWrap* for the keyboard. The effective group is reported to XKB-aware clients in the three reserved bits of the state field of core `KeyPress` and `KeyRelease` events.

The server reports compatibility states to any clients that have not issued a successful `KBUseExtension` request. The server computes the compatibility symbol lookup state and the compatibility effective grab state by applying the compatibility modifier map to the corresponding computed XKB states.

The compatibility symbol lookup state is reported to non-XKB clients in the state field of core `KeyPress` and `KeyRelease` events that do not trigger grabs. The compatibility effective grab state is reported to non-XKB clients in the state field of all core protocol events except `KeyPress` and `KeyRelease` that do not activate passive grabs. The server uses the compatibility effective grab state to activate passive grabs requested by non-XKB clients.

Whenever a client updates the keyboard mapping using a core protocol request, the server saves the requested core protocol keyboard mapping and reports it to any clients that issue `GetKeyboardMapping` or `GetModifierMapping` requests. Whenever a client updates the keyboard mapping using XKB requests, the server discards the affected portion of the stored core keyboard description and regenerates it based on the XKB description of the keyboard as follows:

If the XKB keyboard description defines a single keyboard group, the generated core keyboard mapping has two keysyms-per-keycode, otherwise it has four. The first pair of symbols for each key lists the symbols in the first two positions of the first keyboard group. If two groups of symbols are defined for a key, the second pair contains the symbols in the first two positions of the second keyboard group, otherwise both have the value NoSymbol. If any key type has less than two levels, the second position within each group is reported as NoSymbol.

The generated modifier mapping for a key contains all of the modifiers affected by all of the actions associated with the key, remapped according to the modifier compatibility map. If any of the actions associated with a key affect the keyboard group state, any compatibility modifiers that can be affected by keyboard group are reported in the modifier mask. The `SA_ISOLock` action can theoretically affect any modifier, but the modifier map of an `SA_ISOLock` key contains only the modifiers or group state that it sets by default.

The server notifies interested clients of keyboard map changes in one of two ways. It sends `KBMapNotify` to clients that have explicitly selected them and core protocol `MappingNotify` events to clients that have not. If a client selects `KBMapNotify` events, the server stops sending it `MappingNotify` events.

Section 4. Common Types

The request and event definitions in subsequent sections assume the common types defined by the core X protocol with the following changes and additions:

Name	*Value*
BITMASK	As in the specification of the core protocol, the types BITMASK and LISTofVALUE are used to allow the client to specify a subset of a heterogeneous collection of optional arguments. Unlike the core protocol, the length of a list element depends on context and might vary within a single LISTofVALUE.
CHANGEStoFOO	A type name of the form CHANGEStoFOO means a pair of masks which specify changes to a third mask; all three masks contain elements of type FOO. The first mask of a CHANGEStoFOO is called the *effects mask*, the second is called the *values mask*. The mask to be updated is called the *destination mask*. An argument of type CHANGEStoFOO is applied to a destination mask as follows: Any elements present in both the effects and values masks are added to the destination mask. Any elements present only in the effects mask are cleared from the destination mask. Elements that are not present in the effects mask are not changed in the destination map. If any elements are defined in the values mask but not the effects mask, the server generates a `Match` error. The effects, values and destination masks must all be the same size, but depend on the type of their elements. The representation of the masks will typically contain more bits than there are defined arguments; the unused bits in all masks must be zero or the server generates a `Value` error.
KB_DEVICESPEC	8 bit unsigned integer or `kbUseCoreKeyboard`
KB_EVENTS	{ `kbMapNotify`, `kbStateNotify`, `kbControlsNotify`, `kbIndicatorStateNotify`, `kbIndicatorMapNotify`, `kbNamesNotify`, `kbCompatMapNotify`, `kbAlternateSymsNotify`, `kbBellNotify`, `kbActionMessage`, `kbSlowKeyNotify` }
KB_GROUP	{ 0..7 }
KB_VIRTUALMOD	{ 0..15 } or `kbNoModifier`
KB_INDICATOR	[0..31 } or `kbNoIndicator`
KB_GROUPINFO	{ groupsWrap: BOOL numGroups: 0..8 }

Name	*Value*
KB_STATEMASK	KEYBUTMASK or SETofKB_GROUP
KB_MODMASK	{ mask: KEYMASK realMods: KEYMASK vmods: CARD16 }
KB_STATEPARTS	{ `kbModifierState`, `kbModifierBase`, `kbModifierLatch`, `kbModifierLock`, `kbGroupState`, `kbGroupBase`, `kbGroupLatch`, `kbGroupLock`, `kbCompatMask` }
KB_ACCESSX	{ `kbRepeatKeys`, `kbSlowKeys`, `kbBounceKeys`, `kbStickyKeys`, `kbMouseKeys`, `kbMouseKeysAccel`, `kbAccessXKeys`, `kbAccessXTimeout`, `kbAccessXFeedback` }
KB_CONTROLS	{ `kbGroupsWrap`, `kbAudibleBell`, `kbAutoAutorepeat`, `kbServerInternalMods`, `kbIgnoreLockMods` } or KB_ACCESSX
KB_SLOWKEYEV	{ `kbSK_Press`, `kbSK_Accept`, `kbSK_Reject`, `kbSK_Release` }
KB_MAPPARTS	{ `kbTypes`, `kbKeySyms`, `kbKeyActions`, `kbKeyBehaviors`, `kbVirtualMods`, `kbExplicitComponents` }
KB_COMPATPARTS	{ `kbModCompat`, `kbVirtualModCompat`, `kbSymInterpret` }
KB_NAMES	{ `kbKeycodesName`, `kbGeometryName`, `kbSymbolsName`, `kbSemanticsName`, `kbKeyNames`, `kbKeyTypeNames`, `kbKTLevelNames`, `kbRGNames`, `kbIndicatorNames`, `kbPhysicalNames`, `kbModifierNames`, `kbVirtualModNames`, `kbCharSets` }
KB_EXPLICIT	{ `kbExplicitKeyType`, `kbExplicitInterpret`, `kbExplicitAutorepeat`, `kbExplicitBehavior` }
KB_SIMATCH	{ `NoneOf`, `AnyOfOrNone`, `AnyOf`, `AllOf`, `Exactly` }

Section 5. Errors

The X Keyboard Extension defines a single error, `Keyboard`, which indicates that some request specified an illegal device identifier or an extension device that is not a member of KEYCLASS. Any request with an argument of type KB_DEVICESPEC can cause `Keyboard` errors if an illegal device is specified.

Section 6. Requests

KBUseExtension

wantedMajor, wantedMinor: CARD16
→
supported: BOOL
serverMajor, serverMinor: CARD16

This request enables XKB extension capabilities for the client that issues the request; the *wantedMajor* and *wantedMinor* fields specify the extension version in use by the requesting client. The *supported* field is `True` if the server supports a compatible version, `False` otherwise. The *serverMajor* and *serverMinor* fields return the actual version supported by the server.

Until a client explicitly requests the XKB extension, an XKB capable server reports "compatibility state" in core protocol events and requests. Once a client asks for XKB extension semantics by issuing this request, the server reportss the extended XKB keyboard state in core protocol events and requests.

Clients must issue a `KBUseExtension` request before using any other extension requests.

KBSelectEvents

deviceSpec:KB_DEVICESPEC
affectMap, map: CHANGEStoKB_MAPPARTS
affectWhich: SETofKB_EVENTS
clear: SETofKB_EVENTS
selectAll: SETofKB_EVENTS
masks: LISTofVALUE

Errors: `Keyboard`, `Value`

This request updates the event masks of the keyboard indicated by *deviceSpec* for this client. The event mask for map components that are set in *affectMap* are set to the corresponding value from the *map* field. The *affectWhich* field specifies additional XKB event masks to be changed, *clear* specifies which of those event masks should be cleared and *selectAll* specifies those for which all possible details should be selected. The *masks* field contains changes to the details of any event masks that are specified in *affectWhich* but which do not appear in either of *clear* or *selectAll*.

The server sends a BadMatch error if any event types are present in *clear* or *selectAll* but not in *affectWhich*.

Each entry in *masks* consists of an "affects" mask and a "values" mask whose size and legal contents are determined by the type of event for which they specify details. This table lists the type and size in bytes of the details masks for all XKB events:

Event Type	*Size*	*Details Type*
`KBStateNotify`	2	`CHANGEStoKB_STATEPARTS`

Event Type	*Size*	*Details Type*
KBControlsNotify	4	CHANGEStoKB_CONTROLS
KBIndicatorNotify	4	CHANGEStoINDICATOR
KBIndicatorNotify	4	CHANGEStoINDICATOR
KBNamesNotify	2	CHANGEStoKB_NAMES
KBCompatMapNotify	1	CHANGEStoKB_COMPATMAP
KBAlternateSymNotify	1	CHANGEStoBOOL
KBBellNotify	1	CHANGEStoBOOL
KBActionMessage	1	CHANGEStoBOOL
KBSlowKeyNotify	1	CHANGEStoKB_SLOWKEYEV

For each entry in *masks*, the affects mask specifies which of the details are to be changed and the values mask specifies new values for the changed fields.

KBSendEvent

destination: WINDOW or `PointerWindow` or `InputFocus`
propagate: BOOL
synthesizeClick: BOOL
event-mask: SETofEVENT
event: <normal-event-format>

Errors: `Value`, `Window`

The core `SendEvent` request does not interpret the events it sends; the `KBSendEvent` request modifies the state field of core protocol and input extension events that report state and optionally synthesizes a key click. The state field of any such request is assumed to be an XKB keyboard state, including a keyboard group specification. If the event is sent to an XKB-aware client, the state field is not changed; if the event is to be sent to a non-XKB client, the server applies the modifier compatibility mapping to the event state and group before sending the event.

The server causes an audible key click if all of the following conditions are met:

- the *synthesizeClick* field is `True`
- key click is enabled for the keyboard in question
- the event is of type `KeyPress` or `DeviceKeyPress`
- the server is able to generate a synthetic audible key click

If a server is unable to synthesize key clicks, it is free to ignore the *synthesizeClick* field.

Although it does modify the keyboard state as appropriate, the server does not apply keyboard controls, behavior or key actions to sent key events.

KBBell

deviceSpec: KB_DEVICESPEC
bellClass, bellID: CARD8
percent: INT8
override: BOOL
name: STRING8
window: WINDOW

Errors: `Keyboard`, `Value`

If audible bells are enabled or *override* is `True`, this request sounds the bell specified by *bellClass* and *bellID* on the device specified by *deviceSpec* at the volume specified by *percent*. This request causes a `KBBellNotify` event which contains all of the information about the requested bell, including the otherwise uninterpreted symbolic name specified by *name*. `KBBellNotify` events are generated whether or not the bell was actually rung.

`KBBellNotify` events generated in response to core protocol or X input extension bell requests always report the empty string as their name and `None` as their window.

The *bellClass*, *bellID*, and *percent* fields are interpreted as for the X input extension `DeviceBell` request. The *name* must be a valid atom or `None`; *window* must be a valid window ID or `None`. If any are illegal, this request reports a `Value` error.

KBGetState

deviceSpec: KB_DEVICESPEC
→
deviceID: CARD8
mods, baseMods, latchedMods, lockedMods: KEYMASK
group, baseGroup, latchedGroup, lockedGroup: KB_GROUP
compatState: KEYMASK

Errors: `Keyboard`

This request returns a detailed description of the current state of the keyboard specified by *deviceSpec*.

The *deviceID* return value contains the input extension identifier for the specified device, or `0` if the server does not support the input extension.

The *baseMods* return value reports the modifiers that are set because one or more modifier keys are logically down. The *latchedMods* and *lockedMods* return values report the modifiers that are latched or locked respectively. The *mods* return value reports the effective modifier mask which results from the current combination of base, latched and locked modifiers.

The *baseGroup* return value reports the group state selected by group shift keys that are logically down. The *latchedGroup* and *lockedGroup* return values detail the effects of latching or locking group shift keys and `KBLatchLockState` requests. The *group* return value reports the effective keyboard group which results from the current combination of base, latched and locked group values.

The compatState return value reports the compatibility state that results from applying the compatibility modifier mapping to the effective keyboard group and modifier state.

KBLatchLockState

deviceSpec: KB_DEVICESPEC
affectModLocks, modLocks: CHANGEStoKEYMASK
affectGroupLock: BOOL
groupLock: KB_GROUP
affectModLatches,modLatches: CHANGEStoKEYMASK
affectGroupLatch: KB_GROUP
groupLatch: INT8

Errors: `Keyboard`, `Value`

This request locks or latches keyboard modifiers and group state for the device specified by *deviceSpec*.

The *affectModLocks* and *modLocks* masks are applied to the locked modifiers. If *lockGroup* is `True`, the locked keyboard group is set to the group specified by *groupLock*.

The *affectModLatches* and *modLatches* change masks are applied to the latched state of the keyboard modifiers. If *latchGroup* is `True`, the latched keyboard group is set to the group specified by *groupLatch*.

If either group specification exceeds the maximum number of groups permitted for the specified keyboard, they are wrapped or truncated back into range as specified by the *global groupsWrap* flag. No error results from an out-of-range group specification.

After changing the locked and latched modifiers and groups as specified, the X server recalculates the effective and compatibility keyboard state and generates `KBStateNotify` events as appropriate if any state components have changed. Changing the keyboard state might also change indicators which can cause `KBIndicatorNotify` events.

KBGetControls

deviceSpec: KB_DEVICESPEC
→
deviceID: CARD8
numGroups: CARD8
internalMods, ignoreLockMods: KB_MODMASK
enabledControls: SETofKB_CONTROLS
repeatDelay, repeatInterval: CARD16
slowKeysDelay,debounceDelay: CARD16
mouseKeysDelay, mouseKeysTimeToMax, mouseKeysInterval: CARD16
mouseKeysMaxSpeed, mouseKeysCurve: CARD16
mouseKeysDfltBtn: BUTTON
accessXTimeout: CARD16
accessXTimeoutMask: KB_ACCESSX

Errors: `Keyboard`

This request returns the status of all controls for the keyboard specified by *deviceSpec*.

On return, the *deviceID* specifies the identifier of the requested device or zero if the server does not support the input extension.

The *numGroups* return value reports the current number of groups defined for the keyboard.

The *internalMods* and *ignoreLockMods* masks report the current value of the corresponding special modifier masks. Like all elements of type KB_MODMASK, each contains a mask of real modifiers (of type KEYMASK), a mask of virtual modifiers (of type SETofVIRTUALMOD) and the effective set of modifiers which results from applying the current virtual modifier definitions to the virtual modifier mask and adding the result tp the real modfier mask.

The *enabledControls* mask reports the global controls that are set. The *repeatDelay*, *repeatInterval*, *slowKeysDelay* and *debounceDelay* fields report the current values of the for the autorepeat delay, autorepeat interval, slow keys delay and bounce keys timeout, respectively. All of these times are reported in milliseconds.

If mouse keys are active, pressing and holding a key which cases an `SA_AccelPtr` action causes core pointer events as follows:

The first motion event moves the pointer according to the deltas specified by the the activated action and is generated immediately. The *mouseKeysDelay* specifies the time in milliseconds between the first artificial pointer event and the second; the *mouseKeysInterval* specifies the time between all subsequent artificial pointer events until the key is released. The *mouseKeysTimeToMax* specifies the total number of repeated events before the pointer is travelling at its maximum speed (determined by multiplying the *mouseKeysMaxSpeed* by the deltas specified by the `SA_AccelPtr` action). The *mouseKeysCurve* determines the rate at which the pointer delta goes from minimum to maximum speed. The initial event always moves the pointer the minimum possible distance; after *mouseKeysTimeToMax* events, all events move the pointer the maximum distance. The delta for intermediate event *n* is:

$$\text{delta} = \text{action\_delta} \times n \left(\frac{\text{max\_speed}}{\text{time\_to\_max}^{1 + \frac{\text{curve}}{1000}}} \right)$$

If the the action delta is non-zero, the resulting delta must also be non-zero. If the action delta is positive, the result delta is rounded up, otherwise it is rounded down.

This definition of mouse keys curve allows you to vary the mouse response: a curve of zero approximates a linear increase from minimum to maximum acceleration; increasing the curve value causes longer periods of minimum or low acceleration followed by a rapid increase to maximum acceleration (behaving more like an absolute threshhold as curve increases).

The *mouseKeysDfltBtn* return value reports the pointer button for which events are synthesized by the mouse keys server actions by default.

The *accessXTimeout* return value reports the length of time, in seconds, that the keyboard must remain idle before the the server disables any controls specified in *accessXMask*. An *accessXTimeout* of `0` or empty *accessXMask* indicates that AccessX controls are not automatically deactivated.

KBSetControls

deviceSpec: KB_DEVICESPEC
affectInternalRealMods, internalRealMods: CHANGEStoKEYMASK
affectInternalVirtualMods, internalVirtualMods: CHANGEStoVIRTUALMOD
affectIgnoreLockRealMods, ignoreLockRealMods: CHANGEStoKEYMASK
affectIgnoreLockVirtualMods,ignoreLockVirtualMods: CHANGEStoVIRTUALMOD
affectEnabled, enabled: CHANGEStoKB_ACCESSX
changeControls: SETofKB_CONTROLS
repeatDelay, repeatInterval: CARD16
slowKeysDelay: CARD16
debounceDelay: CARD16
mouseKeysDelay, mouseKeysInterval: CARD16
mouseKeysMaxSpeed, mouseKeysTimeToMax, mouseKeysCurve: CARD16
mouseKeysDfltBtn: BUTTON
accessXTimeout: CARD16
accessXTimeoutMask: KB_ACCESSXMASK

Errors: `Keyboard`, `Value`

This request changes keyboard controls for the keyboard specified by *deviceSpec*.

Controls specified in both *affectEnabled* and *enabled* are turned on; controls specfied in *affectEnabled* alone are turned off. Controls not specified in either are not changed. The server generates a `Match` error if any controls specified in enabled are not also present in *affectEnabled*.

Both the *affectInternalRealMods* and *internalRealMods* change masks and the *affectInternalVirtualMods* and *internalVirtualMods* change masks are applied to the internal modifiers of the keyboard. If either mask changes, the server recomputes the effective internal modifier mask.

The *affectIgnoreLockRealMods*, *ignoreLockRealMods*, *affectIgnoreLockVirtualMods* and *ignoreLockVirtualMods* change masks are applied to the ignore locks modifiers of the keyboard. If either mask changes, the server recomputes the effective ignore locks mask.

The *changeControls* field specifies which of the other request fields should be used to change the values of the corresponding keyboard controls. This table lists the controls that can be included in *changeControls* and the fields they affect:

Control	*Fields*
`RepeatKeys`	repeatDelay, repeatInterval
`SlowKeys`	slowKeysDelay
`BounceKeys`	debounceDelay
`MouseKeys`	mouseKeysDfltBtn
`MouseKeysAccel`	mouseKeysDelay, mouseKeysInterval, mouseKeysMaxSpeed, mouseKeysTimeToMax, mouseKeysCurve
`AccessXTimeout`	accessXTimeout, accessXTimeoutMask

This request can cause `KBControlsNotify` events.

KBGetMap

deviceSpec: KB_DEVICESPEC
full, partial: SETofKB_MAPPARTS
firstType, nTypes: CARD8
firstKeySym, nKeySyms: KEYCODE
firstKeyAction, nKeyActions: KEYCODE
firstKeyBehavior, nKeyBehaviors: KEYCODE
virtualMods: SETofVIRTUALMOD
firstKeyExplicit, nKeyExplicit: KEYCODE

→

deviceID: CARD8
minKeyCode, maxKeyCode: KEYCODE
totalKeyTypes: CARD8
valueMask: BITMASK
valueList: LISTofVALUE
Errors: `Keyboard`, `Value`, `Match`, `Alloc`

This request returns the indicated components of the server and client maps of the keyboard specified by *deviceSpec*. The *full* mask specifies the map components to be returned in full; the *partial* mask specifies the components for which some subset of the legal elements are to be returned. The server returns a `Match` error if any component is specified in both *full* and *partial*.

If some counted map component is specified in *partial*, the corresponding first*Element* and n*Element*s fields must specify legal elements:

Partial Value	*Minimum First*	*Maximum Number*
`kbTypes`	0	nKeyTypes-firstKeyType
`kbKeySymMaps`	minKeyCode	maxKeyCode-firstKeySymMap+1
`kbKeyActions`	minKeyCode	maxKeyCode-firstKeyAction+1
`kbKeyBehaviors`	minKeyCode	maxKeyCode-firstKeyBehavior+1
`kbExplicitComponents`	minKeyCode	maxKeyCode=firstKeyExplicit+1

The nKeyTypes, minKeyCode and maxKeyCode values are taken from the keyboard description, the first*Element* values are taken from the request. Requesting part of a component and specifying elements outside of these legal ranges causes a `Value` error. The first*Element* and n*Element*s fields of any component not specified in partial are ignored.

The first and length specifications for any component specified in the full mask are taken to be the minimum and maximum legal values, respectively. For example, if the full mask contains `KBKeySymMaps`, this request returns `maxKeyCode-minKeyCode+1` key symbol mappings starting from element `minKeyCode`.

The *virtualMods* field specifies the virtual modifiers for which bindings should be returned.

On successful return, the *deviceID* field returns the input extension identifier of the specified device or zero if the server does not support the input extension. The *minKeyCode* and *maxKeyCode* return values report the minimum and maximum legal keycodes for the specified device.

The *valueMask* and *valueList* fields specify the client components for which some elements are being returned.

Component	*Type*
`kbTypes`	LISTofKB_KEYTYPE
`kbKeySymMaps`	LISTofKB_KEYSYMMAP
`kbKeyActions`	LISTofLISTofKB_ACTION
`kbKeyBehaviors`	LISTofKB_BEHAVIOR_VALUE where KB_BEHAVIOR_VALUE{ key: KEYCODE, behavior: KB_BEHAVIOR }
`kbVirtualMods`	LISTofKB_VMOD_VALUE where: KB_VMOD_VALUE { vmod: KB_VIRTUALMOD, mod: KB_MODIFIER }
`kbExplicitComponents`	LISTofKB_EXPLICIT_VALUE where KB_EXPLICIT_VALUE { key: KEYCODE, explicit: SETofKB_EXPLICIT }

The lists of key types, symbols maps and actions return all elements in the requested range. The listsof key behaviors, virtual mods and explicit components report only those elements in the requested range that have "unusual" values. The "usual" keyb behavior has type `KB_Default`. Virtual modifiers are usually not bound to a modifiers and a keys usually have no explicit components.

The *totalKeyTypes* return value reports the total number of key types defined for the keyboard in question.

KBSetMap

deviceSpec: KB_DEVICESPEC
resize: { `kbKeyTypes` }
virtualMods: SETofKB_VIRTUALMOD
valueMask: KB_MAPMASK
valueList: LISTofVALUE

Errors: `Keyboard`, `Value`, `Match`, `Alloc`

This request changes the indicated parts of the keyboard specified by *deviceSpec*. The effect of a key release is independent of the keyboard mapping at the time of the release, so this request succeeds regardless of the logical state of the modifier keys.

The *valueMask* and *valueList* fields specify the keyboard map components to be changed: The values and types are the same as for `kbGetMap`. Each element of *valueList* replaces a subset of one of the components of the keyboard map. The indices of all elements of `kbKeySymMaps`, `kbKeyActions,` `kbKeyBehaviors` and `kbEx-`

plicitComponents to be changed must be in the range of legal keycodes for the device in question or a Value error results.

If kbKeyTypes is set in the resize field, the server resizes the list of key types so that the last key type specified by this request is the last element in the list. An attempt to enlarge the list of key types causes a Value error unless the appropriate bit of the *resize* field is set.

The index of the first changed key type must be less than or equal to the unmodified length of the list of key types or a Value error results. The keyboard map that would result from applying this request must contain only legal values and must be internally consistent, or an error results.

For each result key type:
- Group width must be greater than zero or a Value error results.
- All map elements must be less than the group width or a Match error results.
- All elements of the preserve map must include only modifiers that also in the mask of the key type, or a Match error results.
- If the key type name is ONE_LEVEL, group width must be one (else a Match error).
- If the key type name is TWO_LEVEL or KEYPAD, group width must be two or a Match error results.

For each key in the result key symbol map:
- The key type index must specify a legal result key type or a Match error results.
- The number of groups must be in the range 0..8 or a Value error results.
- The number of symbols must equal the number of groups times the *groupWidth* of the result key type or a Match error results.

For each key in the result key actions:
- The key must have no actions or the number of actions must equal the number of symbols assigned to the key or a Match error results.
- All actions assigned to a key must be legal or a Value error results. All private actions are legal; servers should treat any unknown private actions as SA_NoAction.

The specified behavior of each resulting key behavior must be legal or a Value error results. All private behaviors are legal; servers should treat any unknown private behavior as KB_Default.

Each resulting virtual modifier must be bound to a legal real modifier or have the value kbNoModifier or a Value error results.

All of the unused bits of any resulting set of explicit components for a key must be zero or a Value error results.

If the resulting keyboard map is legal, the server updates the keyboard map. Changes to some keyboard components have indirect effects on others:

If the group width of an existing key type is changed, the list of symbols associated with any keys of the changed type is resized accordingly. If the list increases in size, any unspecified new symbols are initialized to NoSymbol.

If the list of actions associated with a key is not empty, changing the key type of the key resizes the list. Unspecified new actions are usually calculated by applying any keyboard symbol interpretations to the corresponding symbols. If the explicit components map of some key whose type has changed size contains kbExplicitInterpret, the new actions are initialized to SA_NoAction instead.

The number of groups global to the keyboard is always equal to the largest number of groups specified by any of the key symbol maps. Changing the number of groups in one or more key symbol maps may change the number of groups global to the keyboard.

Changing a key behavior from `KB_Lock` to some other behavior causes a synthetic `KeyRelease` event if the key is logically but not physically down at the time of the change.

Symbolic names for newly created key types or levels within a key type are initialized to `None`.

This request sends a `kbMapNotify` event which reflects both explicit and indirect map changes to any interested clients. If any symbolic names are changed, it sends a `kbNamesNotify` reflecting the changes to any interested clients. XKB-unaware clients are notified of keyboard changes via core protocol `MappingNotify` events.

Artificial `KeyPress` and `KeyRelease` events caused by changing key behavior may cause `KeyPress`, `KeyRelease`, `DeviceKeyPress`, `DeviceKeyRelease`, `kbStateNotify` or `kbIndicatorNotify` events.

KBGetCompatMap

deviceSpec: KB_DEVICESPEC
getAllSI: BOOL
firstSI, nSI: CARD16
mods: KEYMASK
vmods: SETofVIRTUALMOD

→

deviceID: CARD8
totalSI: CARD16
siRtrn: LISTofKB_SYMINTERPRET
modsRtrn: LISTofKB_MODCOMPATMAP
vmodsRtrn: LISTofMODCOMPATMAP

where:

KB_SYMINTERPRET:	[levelOneOnly: BOOL, match: KB_SIMATCH, mods: KEYMASK, sym: KEYSYM, action: KB_ACTION, virtualMod: KB_VIRTUALMOD, autorepeat: BOOL, locking: BOOL]
KB_MODCOMPATMAP:	[mods: KEYMASK, groups: SETofKB_GROUP]

Errors: `Keyboard`, `Match`, `Alloc`

This request returns the listed compatibility map components for the keyboard specified by *deviceSpec*.

If *getAllSI* is `False`, *firstSI* and *nSI* specify the subset of the symbol interpretations to be returned; *nSI* must be greater than 0 and all of the elements specified by *firstSI* and *nSI* must be defined or a `Value` error results. If *getAllSI* is `True`, the server ignores *firstSI* and *nSI* and returns all of the symbol interpretations defined for the keyboard.

The *mods* mask specifies the modifiers for which compatibility maps are to be returned. If any modifier is set in *mods*, the corresponding modifier compatibility map is returned.

The *vmods* mask specifies the virtual modifiers for which compatibility maps are to be returned. This request returns modifier compatibility maps for any virtual modifiers specified in vmods.

On return, *deviceID* reports the input extension identifier of the keyboard device or 0 if the server does not support the input extension. The *totalSI* return value reports the total number of symbol interpretations defined for the keyboard, even if the request returns only a subset of them.

The `symsRtrn`, `modsRtrn` and `vmodsRtrn` return values contain the definitions of the requested symbol interpretations and modifier compatibility maps. The `modsRtrn` and `vmodsRtrn` fields report only “unusual” modifier compatibility maps.

The usual modifier compatibility map for a real modifier matches itself only; it does not match any groups or any other modifiers. The usual modifier compatibility map for a virtual modifier matches no groups or modifiers.

KBSetCompatMap

deviceSpec: KB_DEVICESPEC
recomputeXKBActions: BOOL
truncateSI: BOOL
si: LISTofKB_SYMINTERPRET
mods: LISTofKB_MODCOMPATMAP
vmods: LISTofKB_MODCOMPATMAP

where:

KB_SYMINTERPRET:	<as specified for `KBGetCompatMap`>
KB_MODCOMPATMAP:	<as specified for `KBGetCompatMap`>

Errors: `Keyboard`, `Match`, `Value`, `Alloc`

This request changes a specified subset of the compatibility map of the keyboard indicated by *deviceSpec*. The *si*, *mods* and *vmods* fields contain new definitions for subsets of the symbol interpretatations, real and virtual modifier compatibility mappings.

The index of the first symbol interpretation changed by this request must be less than or equal to the unchanged length of the existing list or a `Value` error results. All changed symbol interpretations must specify legal actions or a `Value` error results.

If the index of the last symbol interpretation changed by this request is greater than or equal to the former length of the list the list of symbol interpretations is increased in size automatically. Otherwise, if *truncateSyms* is `True`, the server truncates its list of symbol interpretations after the last element changed by this request..

If the *recomputeXKBActions* field is `True`, the server regenerates the XKB keyboard map by applying the new symbol interpretations to the XKB key symbol map.

This request causes `kbCompatNotify` events. If *recomputeActions* is specified, this request can also cause `kbMapNotify` or `MappingNotify` events.

KBGetIndicatorState

deviceSpec: KB_DEVICESPEC

→

state: SETofKB_INDICATOR

Errors: `Keyboard`

This request reports the current state of all indicators for the keyboard specified by *deviceSpec*. The least-significant bit of the *state* return value corresponds to indicator 0, the most significant bit to indicator 31; if a bit is set, the corresponding indicator is lit.

KBGetIndicatorMap

deviceSpec: KB_DEVICESPEC
which: SETofKB_INDICATOR

→

deviceID: CARD8
nRealIndicators:CARD8
maps: LISTofKB_INDICATORMAP

where:

KB_INDICATORMAP: [noAutomatic: BOOL,
noExplicit: BOOL,
whichGroups: KB_IMPARTS,
groups: SETofKB_GROUP,
whichMods: KB_IMPARTS,
mods: KB_MODMASK,
ctrls: SETofKB_ACCESSX]

Errors: `Keyboard`, `Value`

This request returns a subset of the indicator maps of the keyboard specified by *deviceSpec*. The *which* field specifies the subset to be returned; each bit set in *which* indicates that the map for the corresponding indicator should be returned.

On a successful return, the *deviceID* field reports the input extension identifier of the keyboard or `0` if the server does not support the input extension. The first *nRealIndicators* indicators correspond to indicators actually present on the keyboard, the rest are virtual indicators. Virtual indicators do not directly cause any visible or audible effect when they change state, but they do cause kb`IndicatorNotify` events.

The *maps* return value reports the requested indicator maps. Each indicator map specifies controls, modifier and group combinations that change the state of the corresponding indicator. The *whichGroups* and *whichMods* masks specifies the keyboard state components to be considered for groups and modifiers respectively.

If the *noExplicit* field of an indicator map is `True`, attempts to change the indicator state using core protocol requests are ignored. If keyboard state changes and the *noAutomatic* field of an indicator map is `False`, the indicator state is determined as follows:

The *groups* mask specifies the groups that affect the indicator; if the bit of *groups* which corresponds to the current keyboard group specified by *whichGroups* is set, the indicator is lit. The *mods* mask specifies the real, virtual and effective modifiers that control the corresponding indicator; if any of the modifiers present in the effective modifier mask are also set in the keyboard state mask specified by *whichMods*, the indicator is lit. The *ctrls* mask specifies the keyboard controls whose enabled state affects the indicator; if any of the controls specified in *ctrls* are enabled, the indicator is lit. If none of *mods*, *groups* or *ctrls* match the current keyboard state, the indicator is set to the state explicitly requested by a client (if any) or is turned off if *noExplicit* is `True`.

The server generates a `KBIndicatorNotify` whenever any indicator changes state.

KBSetIndicatorMap

deviceSpec: KB_DEVICESPEC
which: SETofKB_INDICATOR
maps: LISTofKB_INDICATORMAP

where:

KB_INDICATORMAP: <as specified for `KBGetIndicatorMap`>

Errors: `Keyboard`, `Value`

This request changes a subset of the indicator map of keyboard specified by *deviceSpec*. The *which* field specifies the indicator maps to be changed; the *maps* field contains the new definitions. The *whichGroups* and *whichMods* fields of all new indicator maps must specify only legal state components or a `Value` error results. The *ctrls* field of all new indicator maps must specify only legal controls or a `Value` error results.

The real and virtual modifiers specified by the *mods* field of any new indicator maps are used, but the effective mask is ignored; the server always calculates a correct effective mask based on the virtual modifier bindings in effect at the time of the request.

If successful, the new indicator maps are applied immediately and the server generates an `kbIndicatorMapNotify` event. If any indicators change state as a result of the new maps, the server generates `kbIndicatorStateotify` events as appropriate.

KBGetNames

deviceSpec: KB_DEVICESPEC
which: SETofKB_NAMES
→
deviceID: CARD8
present: SETofKB_NAMES
nTypeRtrn: CARD8
valueMask: BITMASK
valueList: LISTofVALUE

Errors: `Keyboard`, `Value`

This request returns the symbolic names for various components of the keyboard mapping for the device specified by *deviceSpec*. The *which* field specifies the keyboard components for which names are to be returned.

The *deviceID* return value contains the X Input Extension device identifier of the specified device or `0` if the server does not support the input extension. The *valueMask* and *valueList* return values specify the components for which names are being reported.

Component	*Type*
`kbKeycodesName`	ATOM
`kbGeometryName`	ATOM
`kbSymbolsName`	ATOM
`kbSemanticsName`	ATOM
`kbKeyTypeNames`	LISTofATOM
`kbKTLevelNames`	LISTofLISTofATOM
`kbIndicatorNames`	LISTofATOM
`kbModifierNames`	LISTofATOM
`kbKeyNames`	LISTofKB_KEYNAME
`kbRGNames`	LISTofATOM
`kbVirtualModNames`	LISTofATOM
`kbCharSets`	LISTofATOM
`kbPhysicalNames`	LISTofATOM

The *nKeyTypesRtrn* field indicates the total number of key types defined for the keyboard. The returned names, if any, for key types and key type levels have *nKeyTypesRtrn* elements. Each element of the list of key type level names is a list of ATOMs with as many elements as the corresponding key type has shift levels.

Values for most names are returned only if they are defined (i.e. are not `None`). The only exceptions are `kbKTLevelNames` and `kbKeyNames`. If any of the levels in a key type have a valid name, names are returned for all levels; even those whose name is `None`. Even if none of the levels in a key type have a valid name, the *valueList* contains an entry for the key type (which consists of a length 0 list of ATOMs). If key names are requested and any keys have non-zero names (any of the four bytes in the name are non-zero), the server returns the names of all keyboard keys.

KBSetNames

deviceSpec: KB_DEVICESPEC
valueMask: BITMASK
valueList: LISTofVALUE

Errors: `Keyboard`, `Atom`, `Value`, `Match`, `Alloc`

This request changes the symbolic names for the requested components of the keyboard specified by *deviceSpec*. All new name definitions of type ATOM must be legal atoms or `None`, or the server sends an `Atom` error.

The *valueMask* and *valueList* fields specify the components to be changed; value and type are as specified for the `kbGetNames` request. If `kbKTLevelNames` is set in *valueMask*, the largest element in the list of key type level names must be less than the number of key types defined for the keyboard or a `Value` error results. The number of level names defined by each entry in *levelNames* must exactly match the *groupWidth* of the corresponding key type or a `Match` error occurs.

The first three key types for any keyboard must have the names ONE_LEVEL, TWO_LEVEL and KEYPAD, respectively and no other type may have one of these names. Attemps to assign names otherwise cause a `Value` error.

This request can cause `kbNamesNotify` events.

KBListAlternateSyms

deviceSpec: KB_DEVICESPEC
name: ATOM or `None`
charset: ATOM or `None`

→

deviceID: CARD8
nAlternateSyms: CARD8
altSymSets: LISTofKB_ALTSYMINFO

where:

KB_ALTSYMINFO:	[id: CARD8, name: ATOM, charSets: LISTofATOM]

Errors: `Keyboard`, `Atom`

This request reports the names of the alternate symbol sets defined for the keyboard specified by *deviceSpec*. The *name* and *charset* fields may be combined to select alternate symbols appropriate for a specific application or locale. If the *name* field has a value other than `None`, only alternate symbol sets whose names match are returned. If the *charset* field is not `None`, only alternate symbol sets whose list of character sets contains a matching entry are returned. If either field contains anything but `None` or a legal atom, an `Atom` error results.

The *deviceID* reports the input extension id of the keyboard or 0 if the server does not support the input extension. The *nAlternateSyms* return value reports the number of alternate symbol sets that match the name and character set. If any matching alternate symbol sets are found, the *altSymSets* return value reports the id, name and complete list of charSets for the matching symbol sets. The id is used to read an alternate symbol set from the server.

KBGetAlternateSyms

deviceSpec: KB_DEVICESPEC
id: CARD8
firstKey, nKeys: CARD8
→
deviceID: CARD8
id: CARD8
name: ATOM
charSets: LISTofATOM
totalSyms: CARD16
altSyms: LISTofKB_KEYSYMMAP

Errors: `Keyboard`, `Value`, `Alloc`

This request returns a description of one alternate symbol set from the keyboard specified by *deviceSpec*. The id field specifies the alternate symbol map to be returned and must specify a legal alternate symbol map or a `Value` error results. The firstKey and nKeys fields specify the subset of the alternate symbol map to be returned as a range of KEYCODEs; both `firstKey` and `firstKey+nKeys-1` must be in the legal range of keycodes for the keyboard in question or a `Value` error results.

On successful return, the *deviceID* field reports the input extension identifier of the keyboard device or `0` if the server does not support the input extension. The id return value reports the index of the alternate symbol map being returned. The *name* and *charSets* return values report the symbolic name and list of supported character sets for the requested alternate symbol map respectively.

The altSyms return value contains the alternate symbol definitions for the requested keys. If any entry in *altSyms* defines more than one group, clients should use the key type, number of groups and symbols defined by the entry. Otherwise, the entry is ignored and clients should use the corresponding key symbol map from the default keyboard definition.

Note Alternate key symbol maps are used only by clients; the key action to be applied is determined using the key symbol map from the default keyboard definition. Assigning alternate symbol maps to keys that initiate some action is not recommended.

KBSetAlternateSyms

deviceSpec: KB_DEVICESPEC
create: BOOL
replace: CARD8
present: { KB_ASName, KB_ASCharSets, KB_ASKeys }
name: ATOM or None
charSets: LISTofATOM
keys: LISTofKB_SETALTSYMMAP

where:

KB_SETALTSYMMAP: [change, defined: BOOL,
map: KB_KEYSYMMAP]

Errors: Keyboard, Value, Atom

This request creates or changes an alternate symbol set defined for the keyboard specified by *deviceSpec*. If *create* is True, this request creates a new alternate symbol map with the specified definition; otherwise, it replaces the listed components of the existing alternate symbol map specified by *replace*. A Value error results if create is True and *replace* does not specify an existing alternate symbol map.

If KB_ASName is set in present, the *name* field specifies a name for the alternate symbol map. If specified, *name* must be a valid ATOM or None or an Atom error results.

If KB_ASCharSets is set in *present*, the *charSets* field specifies a list of supported character sets for the alternate symbol set. An Atom error results if any element of *charSets* is not a valid ATOM or None. If *create* is False, this request replaces the previous list of supported character sets.

If KB_ASKeys is set in present, *keys* specifies new definitions for a range of keys in the alternate symbol set. If the *change* field of any element of *keys* is True, the corresponding key symbol map from the alternate symbol set is changed to reflect the values specified by *defined* and *map*.

KBGetGeometry

deviceSpec: KB_DEVICESPEC

→

supported: BOOL
deviceID: CARD8
minKeyCode, maxKeyCode: CARD8
name: ATOM
<UNDER CONSTRUCTION>
Errors: Keyboard

This request returns a description of the physical layout of the keyboard specified by *deviceSpec*. The *deviceID* field reports the input extension device ID or 0 if the server does not support the input extension. The *name* field reports the symbolic name for the keyboard geometry (identical to the geometry name reported by the KBGetNames request). The *minKeyCode* and *maxKeyCode* fields report the range of legal keycodes supported by the device.

If *supported* is `False`, the server does not have a description of the keyboard layout and returns only the device id, symbolic name and range of keycodes.

KBSetGeometry

deviceSpec: KB_DEVICESPEC
name: ATOM
<UNDER CONSTRUCTION>
Errors: `Keyboard`, `Atom`, `Value`

This request changes the reported description of the geometry for the keyboard specified by *deviceSpec*. The name field must specify a valid ATOM or have the value `None` or an `Atom` error results.

Section 7. Events

KBStateNotify

time: TIMESTAMP
deviceID: CARD8
mods, baseMods, latchedMods, lockedMods: KEYMASK
compatState: KEYMASK
group, baseGroup, latchedGroup, lockedGroup: CARD8
keycode: CARD8
eventType: CARD8
requestMajor, requestMinor: CARD8
changed: KB_STATEPARTMASK

This event is generated whenever any component of a keyboard state changes. The *deviceID* specifies the keyboard for which state changed and the changed mask specifies the components that have changed. The *mods*, *baseMods*, *latchedMods* and *lockedMods* fields report the current state of all modifier components of the keyboard state. The *group*, *baseGroup*, *latchedGroup*, and *lockedGroup* report the current values of all group components of the keyboard state. The *compatState* field reports the current value of the effective compatibility state, which is computed by applying the keyboard modifier mapping to the XKB effective state and group.

If the keyboard state changed because due to a key event, the *keycode* field reports the key that changed and the *eventType* field reports the type of the event that caused the state change.

If the keyboard state changed due to a request, the *keycode* field is 0 and the *requestMajor* and *requestMinor* fields report the request that caused the state change.

`KBStateNotify` events are reported only to clients that specifically request them. Using the `KBSelectEvents` request, clients can express interest in changes in specific state components.

KBMapNotify

time: TIMESTAMP
deviceID: CARD8
changed, resize: SETofKB_MAPPARTS
firstType, nTypes: CARD8
firstKeySym, nKeySyms: KEYCODE
firstKeyAction, nKeyActions: KEYCODE
firstKeyBehavior, nKeyBehaviors: KEYCODE
firstKeyExplicit, nKeyExplicit: KEYCODE
virtualMods: SETofKB_VIRTUALMOD

This event is generated whenever some aspect of the keyboard map is changed by an XKB or core protocol request. The *deviceID* specifies the keyboard for which some component changed and the present mask specifies the changed components.

Each *firstElement* and n*Element*s pair specifies the elements that have changed in the corresponding keyboard component. If the last element of any resizable map component is one of the changed element, the corresponding bit is set in the resize mask of this event.

The *virtualMods* return value reports virtual modifiers with changed bindings.

By default, `KBMapNotify` events are reported to all clients that have issued a `KBUseExtension` request whenever the key types or key symbol maps changed. Changes in other map components are reported only to clients that explicitly request them.

KBControlsNotify

time: TIMESTAMP
deviceID: CARD8
changedCtrls: SETofKB_CONTROLS
enabledCtrls, enabledCtrlChanges: SETofKB_ACCESSX

This event is generated whenever any of the keyboard controls are changed by an XKB or core protocol request, by the AccessX key sequences or by a key action. The *deviceID* field reports the keyboard for which some control changed. The *changedCtrls* field reports controls for which some configuration value changed (e.g. if autorepeat delay and interval are change, *changedCtrls* include `kbRepeatKeys`). The *enabledCtrlChanges* field reports all controls that were turned on or off and *enabledCtrls* reports the current state of all controls.

This event is not reported by default; clients may express interest in changes to individual controls using the `KBSelectEvents` request.

KBIndicatorStateNotify

time: TIMESTAMP
deviceID: CARD8
stateChanged, state: SETofKB_INDICATOR
mapChanged: SETofKB_INDICATOR

This event is generated whenever any physical or virtual indicator changes state. The *deviceID* field specifies the device on which the change occurred.

The *stateChanged* field lists any indicators whose state have changed; the *state* field reports the current state of all thirty-two keyboard indicators. If any indicator maps have also changed, the *mapChanged* field reports them.

This event is not reported by default. Clients may express interest in changes in the state of individual indicators using the `KBSelectEvents` request.

KBIndicatorMapNotify

time: TIMESTAMP
deviceID: CARD8
stateChanged, state: SETofKB_INDICATOR
mapChanged: SETofKB_INDICATOR

This event is generated whenever any some client changes the mapping of one or more indicators. The *deviceID* field specifies the device on which the change occurred.

The *mapChanged* field lists any indicators whose map has changed. The *state* field reports the current state of all keyboard indicators.

If any indicators have changed state, the *stateChanged* field reports them.

This event is not reported by default. Clients may express interest in changes in the mapping of individual indicators using the `KBSelectEvents` request.

KBBellNotify

time: TIMESTAMP
deviceID: CARD8
bellClass, bellID: CARD8
percent: CARD8
pitch: CARD16
duration: CARD16
name: ATOM
window: WINDOW

This event is generated when some client issues a `Bell`, `DeviceBell`, or `KBBell` request. The *deviceID* field reports the keyboard for which a bell was requested the *bellClass* and *bellID* fields specify the input extension class and identifier of the feedback for which the bell was requested. If the reporting server does not support the input extension, both *bellClass* and *bellID* are 0.

The *percent*, *pitch* and *duration* fields report the volume, tone and duration requested for the bell as specified by the core protocol and input extension.

If the client used the `KBBell` request, the *name* field contains a symbolic name for the bell and the *window* field indicates the window for which the bell was generated. Either field has the value `None` if the client used a core protocol or input extension request or if the client did not supply a name or window.

This event is reported only to clients that specifically request it. Whether or not this event is reported is independent of audible bell keyboard control; `KBBellNotify` events can be reported even if an audible bell is also sounded.

KBSlowKeyNotify

time: TIMESTAMP
deviceID: CARD8
slowKeyState: SETofKB_SLOWKEYEV
keycode: KEYCODE
delay: CARD16

This event is generated whenever a key changes state or the slow keys timer expires while the slow keys control is enabled. The deviceID specifies the keyboard on which the event occurred. The keycode specifies the key for which the event occurred and the delay field reports the current slow keys delay for the keyboard in question. The type field describes the event that occurred and has one of the following values:

Value	***Meaning***
`kbSK_Press`	The user pressed the key specified by keycode.
`kbSK_Accept`	The slow keys timer expired and the key was accepted
`kbSK_Reject`	The user released the key before the timer expired.
`kbSK_Release`	The user released the key after the timer expired.

This event is reported only to clients that specifically request it. Clients can request notification of any combination of press, acceptance, rejection and release using the `KBSelectEvents` request.

KBNamesNotify

time: TIMESTAMP
deviceID: CARD8
changed: KB_NAMESMASK
firstType, nTypes: CARD8
firstLevelName, nLevelNames: CARD8
firstRadioGroup, nRadioGroups: CARD8
changedMods: KEYMASK
changedVirtualMods: SETofKB_VIRTUALMOD
changedIndicators: SETofKB_INDICATOR
firstCharSet, nCharSets: CARD8

This event is generated whenever the symbolic names for any components of the keyboard specified by *deviceID* change. Symbolic names can change when:

- Some client explicitly changes them using `KBSetNames`.
- The list of key types is resized
- The group width of some key type is changed

The *changed* mask lists the components for which some elements have changed. The various first*Element* and n*Element*s fields specify the subset of the corresponding components for which names were changed. If the name of the last element of any resizable component was changed, the corresponding bit of the resize mask is set.

This request is not reported by default. Clients can request notification of changes to the name of individual keyboard components with the `KBSelectEvents` request.

KBCompatMapNotify

time: TIMESTAMP
deviceID: CARD8
changedMods: KEYMASK
changedVMods: SETofKB_VIRTUALMOD
firstSI, nSI: CARD16
nTotalSI: CARD16

This event is generated whenever some client changes the compatibility map of the keyboard specified by *deviceID* using the `KBSetCompatMap` request. The *changed-Mods* mask reports any changed compatibility maps for real modifiers; *changedV-Mods* reports changed maps for virtual modifiers. The *firstSI* and *nSI* fields report a range of changed symbol interpretations. The *nTotalSI* field reports the total number of symbol interpretations defined for the keyboard.

This request is not reported by default. Clients can request notification of changes to any compatibility map component using `KBSelectEvents`.

KBAlternateSymsNotify

time: TIMESTAMP
deviceID: CARD8
setDeleted: BOOL
newSet: BOOL
altSymsID: CARD8
firstKey, nKeys: CARD8

This event is generated whenever some client creates, destroys or changes an alternate symbol set for the keyboard specified by *deviceID*. If the *setDeleted* field is `True`, the set specified by *altSymsID* was removed by some client. Otherwise, the *firstKey* and *nKeys* fields specify the range of keys that were changed in the alternate symbol set. If *newSet* is `True`, the set was created and all keys outside of the range specified by *first-Key* and *nKeys* are taken from the default keyboard layout.

KBActionMessage

time: TIMESTAMP
deviceID: CARD8
keycode: KEYCODE
press: BOOL
keyEventFollows: BOOL
message: LISTofCARD8

This event can be generated when a key event activates an `SA_ActionMessage` action; the exact circumstances under which an action causes an event depend on the arguments to the action and the arguments to the action. See the definition of the `SA_ActionMessage` on page 16 for more details.

The *deviceID* field reports the keyboard from which the event was generated and the *keycode* field reports the key that caused the event. If this event was generated due to a key press, the *press* field is `True`.

If *keyEventFollows* is `True`, the server also generated the core protocol key or input extension device key event which the key transition that caused this event; if *keyEventFollows* is `False`, the `SA_ActionMessage` suppressed the corresponding key or device key event. Unsuppressed key events are sent according to the standard core protocol or input extension focus rules, so there is no guarantee that a client which receives a `KBActionMessage` will also receive the corresponding key event. The XKB extension guarantees only that action message events will precede the corresponding key events and that the two events will have identical time stamps and sequence numbers.

The *message* field contains up to six bytes of data reported by the `SA_ActionMessage`; XKB does not interpret the message in any way.

Appendix A. Protocol Encoding

Not ready yet.

Appendix B. Default Symbol Transformations

Control Modifier

This table lists the decimal value of the standard control characters that correspond to some keysyms for ASCII characters. Control characters for symbols not listed in this table are application-specific.

Keysyms	*Value*	*Keysyms*	*Value*	*Keysyms*	*Value*	*Keysyms*	*Value*
atsign	0	h, H	8	p, P	16	x, X	24
a, A	1	i, I	9	q, Q	17	y, Y	25
b, B	2	j, J	10	r, R	18	z, Z	26
c, C	3	k, K	11	s, S	19	left_bracket	27
d, D	4	l, L	12	t, T	20	backslash	28
e, E	5	m, M	13	u, U	21	right_bracket	29
f, F	6	n, N	14	v, V	22	asciicircum	30
g, G	8	o, O	15	w, W	23	underbar	31

Locale-Insensitive Capitalization

List the default capitalization rules, basically codify the table in Xlib today. Necessary so the C locale isn't too stupid.

Appendix C. Canonical Key Types

Describe ONE_LEVEL, TWO_LEVEL and KEYPAD key types.

Appendix D. New KeySyms

Byte	*Byte*	*Code*	*Character*	*Name*
				ISO LOCK
				ISO LATCHING LEVEL TWO SHIFT
				ISO LEVEL THREE SHIFT
				ISO LATCHING LEVEL THREE SHIFT
				ISO LEVEL THREE SHIFT LOCK
				ISO GROUP SHIFT
				ISO GROUP SHIFT LOCK
				ISO LATCHING GROUP SHIFT
				ISO NEXT GROUP
				ISO PREVIOUS GROUP
				ISO FIRST GROUP
				ISO LAST GROUP
				MOUSE LEFT
				MOUSE RIGHT
				MOUSE UP
				MOUSE DOWN
				DEFAULT MOUSE BUTTON
				MOUSE BUTTON ONE
				MOUSE BUTTON TWO
				MOUSE BUTTON THREE
				MOUSE BUTTON FOUR
				MOUSE BUTTON FIVE
				DEFAULT MOUSE BUTTON DOUBLE CLICK
				MOUSE BUTTON ONE DOUBLE CLICK
				MOUSE BUTTON TWO DOUBLE CLICK
				MOUSE BUTTON THREE DOUBLE CLICK
				MOUSE BUTTON FOUR DOUBLE CLICK
				MOUSE BUTTON FIVE DOUBLE CLICK
				DRAG DEFAULT MOUSE BUTTON
				DRAG MOUSE BUTTON ONE
				DRAG MOUSE BUTTON TWO
				DRAG MOUSE BUTTON THREE
				DRAG MOUSE BUTTON FOUR
				DRAG MOUSE BUTTON FIVE
				PREVIOUS SCREEN
				NEXT SCREEN
				FIRST SCREEN

Byte	*Byte*	*Code*	*Character*	*Name*
				LAST SCREEN
				TERMINATE SERVER
				LEFT TAB
				MOVE LINE UP
				MOVE LINE DOWN
				PARTIAL LINE UP
				PARTIAL LINE DOWN
				PARTIAL SPACE LEFT
				PARTIAL SPACE RIGHT
				SET MARGIN LEFT
				SET MARGIN RIGHT
				RELEASE MARGIN LEFT
				RELEASE MARGIN RIGHT
				RELEASE MARGIN LEFT AND RIGHT
				FAST CURSOR LEFT
				FAST CURSOR RIGHT
				FAST CURSOR UP
				FAST CURSOR DOWN
				CONTINUOUS UNDERLINE
				DISCONTINUOUS UNDERLINE
				EMPHASIZE
				CENTER OBJECT
				ISO_ENTER
				DEAD GRAVE ACCENT
				DEAD ACUTE ACCENT
				DEAD CIRCUMFLEX
				DEAD TILDE
				DEAD MACRON
				DEAD BREVE
				DEAD ABOVEDOT
				DEAD DIAERESIS
				DEAD ABOVERING
				DEAD DOUBLE ACUTE ACCENT
				DEAD CARON
				DEAD CEDILLA
				DEAD OGONEK
				DEAD IOTA
				DEAD VOICED SOUND
				DEAD SEMI VOICED SOUND