The X Keyboard Extension: Protocol Specification

X Consortium Standard

Erik Fortune

The X Keyboard Extension: Protocol Specification: X Consortium Standard

Erik Fortune

X Version 11, Release 7.7

Version 1.0

Copyright © 1995, 1996 X Consortium Inc., Silicon Graphics Inc., Hewlett-Packard Company, Digital Equipment Corporation

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

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

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

Except as contained in this notice, the names of the X Consortium, Silicon Graphics Inc., Hewlett-Packard Company, and Digital Equipment Corporation shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization.

Table of Contents

Acknowledgments

I am grateful for all of the comments and suggestions I have received over the years. I could not possibly list everyone who has helped, but a few people have gone well above and beyond the call of duty and simply must be listed here.

My managers here at SGI, Tom Paquin (now at Netscape) and Gianni Mariani were wonderful. Rather than insisting on some relatively quick, specialized proprietary solution to the keyboard problems we were having, both Tom and Gianni understood the importance of solving them in a general way and for the community as a whole. That was a difficult position to take and it was even harder to maintain when the scope of the project expanded beyond anything we imagined was possible. Gianni and Tom were unflagging in their support and their desire to “do the right thing” despite the schedule and budget pressure that intervened from time to time.

Will Walker, at Digital Equipment Corporation, has been a longtime supporter of XKB. His help and input was essential to ensure that the extension as a whole fits and works together well. His focus was AccessX but the entire extension has benefited from his input and hard work. Without his unflagging good cheer and willingness to lend a hand, XKB would not be where it is today.

Matt Landau, at the X Consortium, stood behind XKB during some tough spots in the release and standardization process. Without Matt’s support, XKB would likely not be a standard for a long time to come. When it became clear that we had too much to do for the amount of time we had remaining, Matt did a fantastic job of finding people to help finish the work needed for standardization.

One of those people was George Sachs, at Hewlett-Packard, who jumped in to help out. His help was essential in getting the extension into this release. Another was Donna Converse, who helped figure out how to explain all of this stuff to someone who hadn’t had their head buried in it for years.

Amber Benson and Gary Aitken were simply phenomenal. They jumped into a huge and complicated project with good cheer and unbelievable energy. They were “up to speed” and contributing within days. I stand in awe of the amount that they managed to achieve in such a short time. Thanks to Gary and Amber, the XKB library specification is a work of art and a thousand times easier to use and more useful than it would otherwise be.

I truly cannot express my gratitude to all of you, without whom this would not have been possible.

Erik Fortune

Silicon Graphics, Inc.

5 February 1996

Chapter 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* , *Control* and *Lock* modifiers and the modifiers bound to the *Mode_switch* or *Num_Lock* keysyms interact to generate keysyms and characters. The core protocol also 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 keyboard overlays, PC-style break keys or keyboards that comply with ISO9995 or a host of other national and international standards.
- Use of a modifier to specify a second keyboard group has side-effects that wreak havoc with client grabs and X toolkit translations and limit us to two keyboard groups.
- 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 incompatibilities between X servers with no way for clients to detect implementation differences.
- 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.
- 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 using shift 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 mishmash 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. It also includes a number of keyboard controls designed to make keyboards more accessible to people with movement impairments.

The X Keyboard Extension essentially replaces the core protocol definition of a keyboard. The following sections describe the new capabilities of the extension and the effect of the extension on core protocol requests, events and errors.

Conventions and Assumptions

This document uses the syntactic conventions, common types, and errors defined in sections two through four of the specification of the X Window System Protocol.

This document assumes familiarity with the fundamental concepts of X, especially those related to the way that X handles keyboards. Readers who are not familiar with the meaning or use of keycodes, keysyms or modifiers should consult (at least) the first five chapters of the protocol specification of the X Window System before continuing.

Chapter 2. Keyboard State

The core protocol description of keyboard state consists of eight *modifiers* (*Shift* , *Lock* , *Control* , and *Mod1* - *Mod5*). A modifier reports the state of one or modifier keys, which are similar to qualifier keys as defined by the ISO9995 standard:

Qualifier key	A key whose operation has no immediate effect, but which, for as long as it is held down, modifies the effect of other keys. A qualifier key may be, for example, a shift key or a control key.

Whenever a modifier key is physically or logically depressed, the modifier it controls 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 retains the eight "real" modifiers defined by the core protocol but extends the core protocol notion of *keyboard state* to include up to four *keysym groups* , as defined by the ISO9995 standard:

Group:	A logical state of a keyboard providing access to a collection of characters. A group usually contains a set of characters which logically belong together and which may be arranged on several shift levels within that group.

For example, keyboard group can be used to select between multiple alphabets on a single keyboard, or to access less-commonly used symbols within a character set.

Locking and Latching Modifiers and Groups

With the core protocol, there is no way to tell whether a modifier is set due to a lock or because the user is actually holding down a key; this can make for a clumsy user-interface as locked modifiers or group state interfere with accelerators and translations.

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

Fundamental Components of XKB Keyboard State

The fundamental components of XKB keyboard state include:

- 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).

- State of the core pointer buttons.

The latched and locked state of modifiers and groups can be changed in response to keyboard activity or under application control using the *XkbLatchLockState* request. The base modifier, base group and pointer button states always reflect the logical state of the keyboard and pointer and change *only* in response to keyboard or pointer activity.

Computing Effective Modifier and Group

The effective modifiers and group report the cumulative effects of the base, latched and locked modifiers and group respectively, and cannot be directly changed. Note that the effective modifiers and effective group are computed differently.

The effective modifiers are simply the bitwise union of the base, latched and locked modifiers.

The effective group is the arithmetic sum of the base, latched and locked groups. The locked and effective keyboard group must fall in the range *Group1* - *Group4* , so they are adjusted into range as specified by the global *GroupsWrap* control as follows:

- If the *RedirectIntoRange* flag is set, the four least significant bits of the groups wrap control specify the index of a group to which all illegal groups correspond. If the specified group is also out of range, all illegal groups map to *Group1*.
- If the *ClampIntoRange* flag is set, out-of-range groups correspond to the nearest legal group. Effective groups larger than the highest supported group are mapped to the highest supported group; effective groups less than *Group1* are mapped to *Group1* . For example, a key with two groups of symbols uses *Group2* type and symbols if the global effective group is either *Group3* or *Group4*.
- If neither flag is set, group is wrapped into range using integer modulus. For example, a key with two groups of symbols for which groups wrap uses *Group1* symbols if the global effective group is *Group3* or *Group2* symbols if the global effective group is *Group4*.

The base and latched keyboard groups are unrestricted eight-bit integer values and are not affected by the *GroupsWrap* control.

Computing A State Field from an XKB State

Many events report the keyboard state in a single *state* field. Using XKB, a state field combines modifiers, group and the pointer button state into a single sixteen bit value as follows:

- Bits 0 through 7 (the least significant eight bits) of the effective state comprise a mask of type KEYMASK which reports the state modifiers.
- Bits 8 through 12 comprise a mask of type BUTMASK which reports pointer button state.
- Bits 13 and 14 are interpreted as a two-bit unsigned numeric value and report the state keyboard group.

- Bit 15 (the most significant bit) is reserved and must be zero.

It is possible to assemble a state field from any of the components of the XKB keyboard state. For example, the effective keyboard state would be assembled as described above using the effective keyboard group, the effective keyboard modifiers and the pointer button state.

Derived Components of XKB Keyboard State

In addition to the fundamental state components, XKB keeps track of and reports a number of state components which are derived from the fundamental components but stored and reported separately to make it easier to track changes in the keyboard state. These derived components are updated automatically whenever any of the fundamental components change but cannot be changed directly.

The first pair of derived state components control the way that passive grabs are activated and the way that modifiers are reported in core protocol events that report state. The server uses the *ServerInternalModifiers* , *IgnoreLocksModifiers* and *IgnoreGroupLock* controls, described in Server Internal Modifiers and Ignore Locks Behavior, to derive these two states as follows:

- The lookup state is the state used to determine the symbols associated with a key event and consists of the effective state minus any server internal modifiers.
- The grab state is the state used to decide whether a particular event triggers a passive grab and consists of the lookup state minus any members of the ignore locks modifiers that are not either latched or logically depressed. If the ignore group locks control is set, the grab state does not include the effects of any locked groups.

Server Internal Modifiers and Ignore Locks Behavior

The core protocol does not provide any way to exclude certain modifiers from client events, so there is no way to set up a modifier which affects only the server.

The modifiers specified in the mask of the *InternalMods* control 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 affect only the action applied when a key is pressed.

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 another mask which can help avoid some of these problems.

The locked state of the modifiers specified in mask of the *IgnoreLockMods* control is not reported in most core protocol events and is not used to activate grabs. The only core events which include the locked state of the modifiers in the ignore locks mask are key press and release events that do not activate a passive grab and which do not occur while a grab is active. If the *IgnoreGroupLock* control is set, the locked state of the keyboard group is not considered when activating passive grabs.

Without XKB, the passive grab set by a translation (e.g. *Alt<KeyPress>space*) does not trigger if any modifiers other than those specified by the translation are set, with the result that many user interface components do not react when either

Num Lock or when the secondary keyboard group are active. The ignore locks mask and the ignore group locks control make it possible to avoid this behavior without exhaustively grabbing every possible modifier combination.

Compatibility Components of Keyboard State

The core protocol interpretation of keyboard modifiers does not include direct support for multiple groups, so XKB reports the effective keyboard group to XKB-aware clients using some of the reserved bits in the state field of some core protocol events, as described in Computing A State Field from an XKB State.

This modified state field would not be interpreted correctly by XKB-unaware clients, so XKB provides a *group compatibility mapping* (see Group Compatibility Map) which remaps the keyboard group into a core modifier mask that has similar effects, when possible. XKB maintains three compatibility state components that are used to make non-XKB clients work as well as possible:

- The *compatibility state* corresponds to the effective modifier and effective group state.
- The *compatibility lookup state* is the core-protocol equivalent of the lookup state.
- The *compatibility grab state* is the nearest core-protocol equivalent of the grab state.

Compatibility states are essentially the corresponding XKB state, but with keyboard group possibly encoded as one or more modifiers; Group Compatibility Map describes the group compatibility map, which specifies the modifier(s) that correspond to each keyboard group.

The compatibility state reported to XKB-unaware clients for any given core protocol event is computed from the modifier state that XKB-capable clients would see for that same event. For example, if the ignore group locks control is set and group 2 is locked, the modifier bound to *Mode_switch* is not reported in any event except (Device)KeyPress and (Device)KeyRelease events that do not trigger a passive grab.

Note

Referring to clients as "XKB-capable is somewhat misleading in this context. The sample implementation of XKB invisibly extends the X library to use the keyboard extension if it is present. This means that most clients can take advantage of all of XKB without modification, but it also means that the XKB state can be reported to clients that have not explicitly requested the keyboard extension. Clients that *directly* interpret the state field of core protocol events or that interpret the keymap directly may be affected by some of the XKB differences; clients that use library or toolkit routines to interpret keyboard events automatically use all of the XKB features.

XKB-aware clients can query the keyboard state at any time or request immediate notification of a change to any of the fundamental or derived components of the keyboard state.

Chapter 3. Virtual Modifiers

The core protocol specifies that certain keysyms, when bound to modifiers, affect the rules of keycode to keysym interpretation for all keys; for example, when *Num_Lock* is bound to some modifier, that modifier is used to choose shifted or unshifted state for the numeric keypad keys. The core protocol does not provide a convenient way to determine the mapping of modifier bits, in particular *Mod1* through *Mod5* , to keysyms such as *Num_Lock* and *Mode_switch* . Clients must retrieve and search the modifier map to determine the keycodes bound to each modifier, and then retrieve and search the keyboard mapping to determine the keysyms bound to the keycodes. They must repeat this process for all modifiers whenever any part of the modifier mapping is changed.

XKB provides a set of sixteen named virtual modifiers, each of which can be bound to any set of the eight "real" modifiers (*Shift* , *Lock* , *Control* and *Mod1* - *Mod5* as reported in the keyboard state). This makes it easier for applications and keyboard layout designers to specify to the function a modifier key or data structure should fulfill without having to worry about which modifier is bound to a particular keysym.

The use of a single, server-driven mechanism for reporting changes to all data structures makes it easier for clients to stay synchronized. For example, the core protocol specifies a special interpretation for the modifier bound to the *Num_Lock* key. Whenever any keys or modifiers are rebound, every application has to check the keyboard mapping to make sure that the binding for *Num_Lock* has not changed. If *Num_Lock* is remapped when XKB is in use, the keyboard description is automatically updated to reflect the new binding, and clients are notified immediately and explicitly if there is a change they need to consider.

The separation of function from physical modifier bindings also makes it easier to specify more clearly the intent of a binding. X servers do not all assign modifiers the same way — for example, *Num_Lock* might be bound to *Mod2* for one vendor and to *Mod4* for another. This makes it cumbersome to automatically remap the keyboard to a desired configuration without some kind of prior knowledge about the keyboard layout and bindings. With XKB, applications simply use virtual modifiers to specify the behavior they want, without regard for the actual physical bindings in effect.

XKB puts most aspects of the keyboard under user or program control, so it is even more important to clearly and uniformly refer to modifiers by function.

Modifier Definitions

Use an *XKB modifier definition* to specify the modifiers affected by any XKB control or data structure. An XKB modifier definition consists of a set of real modifiers, a set of virtual modifiers, and an effective mask. The mask is derived from the real and virtual modifiers and cannot be explicitly changed — it contains all of the real modifiers specified in the definition *plus* any real modifiers that are bound to the virtual modifiers specified in the definition. For example, this modifier definition specifies the numeric lock modifier if the *Num_Lock* keysym is not bound to any real modifier:

```
{ real_mods= None, virtual_mods= NumLock, mask= None }
```

If we assign *Mod2* to the *Num_Lock* key, the definition changes to:

```
{ real_mods= None, virtual_mods= NumLock, mask= Mod2 }
```

Using this kind of modifier definition makes it easy to specify the desired behavior in such a way that XKB can automatically update all of the data structures that make up a keymap to reflect user or application specified changes in any one aspect of the keymap.

The use of modifier definitions also makes it possible to unambiguously specify the reason that a modifier is of interest. On a system for which the *Alt* and *Meta* keysyms are bound to the same modifier, the following definitions behave identically:

```
{ real_mods= None, virtual_mods= Alt, mask= Mod1 }
{ real_mods= None, virtual_mods= Meta, mask= Mod1 }
```

If we rebind one of the modifiers, the modifier definitions automatically reflect the change:

```
{ real_mods= None, virtual_mods= Alt, mask= Mod1 }
{ real_mods= None, virtual_mods= Meta, mask= Mod4 }
```

Without the level of indirection provided by virtual modifier maps and modifier definitions, we would have no way to tell which of the two definitions is concerned with *Alt* and which is concerned with *Meta*.

Inactive Modifier Definitions

Some XKB structures ignore modifier definitions in which the virtual modifiers are unbound. Consider this example:

```
if ( state matches { Shift } ) Do OneThing;
if ( state matches { Shift+NumLock } ) Do Another;
```

If the *NumLock* virtual modifier is not bound to any real modifiers, these effective masks for these two cases are identical (i.e. they contain only *Shift*). When it is essential to distinguish between *OneThing* and Another, XKB considers only those modifier definitions for which all virtual modifiers are bound.

Virtual Modifier Mapping

XKB maintains a *virtual modifier mapping* , which lists the virtual modifiers associated with each key. The real modifiers bound to a virtual modifier always include all of the modifiers bound to any of the keys that specify that virtual modifier in their virtual modifier mapping.

For example, if *Mod3* is bound to the *Num_Lock* key by the core protocol modifier mapping, and the *NumLock* virtual modifier is bound to they *Num_Lock* key by the virtual modifier mapping, *Mod3* is added to the set of modifiers associated with the *NumLock* virtual modifier.

The virtual modifier mapping is normally updated automatically whenever actions are assigned to keys (see Changing the Keyboard Mapping Using the Core Protocol for details) and few applications should need to change the virtual modifier mapping explicitly.

Chapter 4. Global Keyboard Controls

The X Keyboard Extension supports a number of *global key controls* , which affect the way that XKB handles the keyboard as a whole. Many of these controls make the keyboard more accessible to the physically impaired and are based on the AccessDOS package[1].

The RepeatKeys Control

The core protocol only allows control over whether or not the entire keyboard or individual keys should autorepeat when held down. The *RepeatKeys* control extends this capability by adding control over the delay until a key begins to repeat and the rate at which it repeats. *RepeatKeys* is also coupled with the core autorepeat control; changes to one are always reflected in the other.

The *RepeatKeys* control has two parameters. 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.

The PerKeyRepeat Control

When *RepeatKeys* are active, the *PerKeyRepeat* control specifies whether or not individual keys should autorepeat when held down. XKB provides the *PerKeyRepeat* for convenience only, and it always parallels the *auto-repeats* field of the core protocol *GetKeyboardControl* request — changes to one are always reflected in the other.

Detectable Autorepeat

The X server usually generates both press and release events whenever an autorepeating key is held down. If an XKB-aware client enables the *DetectableAutorepeat* per-client option for a keyboard, the server sends that client a key release event only when the key is *physically* released. For example, holding down a key to generate three characters without detectable autorepeat yields:

```
Press -> Release -> Press -> Release -> Press -> Release
```

If detectable autorepeat is enabled, the client instead receives:

```
Press-> Press -> Press -> Release
```

Note that only clients that request detectable autorepeat are affected; other clients continue to receive both press and release events for autorepeating keys. Also note that support for detectable autorepeat is optional; servers are not required to support detectable autorepeat, but they must correctly report whether or not it is supported.

---

[1] AccessDOS provides access to the DOS operating system for people with physical impairments and was developed by the Trace R&D Center at the University of Wisconsin. For more information on AccessDOS, contact the Trace R&D Center, Waisman Center and Department of Industrial Engineering, University of Wisconsin-Madison WI 53705-2280. Phone: 608-262-6966. e-mail: info@trace.wisc.edu.

Querying and Changing Per-Client Flags describes the *XkbPerClientFlags* request, which reports or changes values for all of the per-client flags, and which lists the per-client flags that are supported.

The SlowKeys Control

Some users often bump keys accidentally while moving their hand or typing stick toward the key they want. Usually, the keys that are bumped accidentally are hit only for a very short period of time. The *SlowKeys* control helps filter these accidental bumps by telling the server to wait a specified period, called the *SlowKeys acceptance delay* , before delivering key events. If the key is released before this period elapses, no key events are generated. The user can then bump any number of keys on their way to the one they want without generating unwanted characters. Once they have reached the key they want, they can then hold it long enough for *SlowKeys* to accept it.

The *SlowKeys* control has one parameter; the *slow keys delay* specifies the length of time, in milliseconds, that a key must be held down before it is accepted.

When *SlowKeys* are active, the X Keyboard Extension reports the initial press, acceptance, rejection or release of any key to interested clients using *AccessXNotify* events. The *AccessXNotify* event is described in more detail in Events.

The BounceKeys Control

Some people with physical impairments accidentally "bounce" on a key when they press it. That is, they press it once, then accidentally press it again immediately. The *BounceKeys* control temporarily disables a key after it has been pressed, effectively "debouncing" the keyboard.

The *BounceKeys* has a single parameter. The *BounceKeys delay* specifies the period of time, in milliseconds, that the key is disabled after it is pressed.

When *BounceKeys* are active, the server reports the acceptance or rejection of any key to interested clients by sending an *AccessXNotify* event. The *AccessXNotify* event is described in more detail in Events.

The StickyKeys Control

Some people find it difficult or impossible to press two keys at once. The *StickyKeys* control makes it easier for them to type by changing the behavior of the modifier keys. When *StickyKeys* are enabled, a modifier is latched when the user presses it just once, so the user can first press a modifier, release it, then press another key. For example, to get an exclamation point (!) on a PC-style keyboard, the user can press the *Shift* key, release it, then press the *1* key.

By default, *StickyKeys* also allows users to lock modifier keys without requiring special locking keys. The user can press a modifier twice in a row to lock it, and then unlock it by pressing it one more time.

Modifiers are automatically unlatched when the user presses a non-modifier key. For instance, to enter the sequence *Shift* + *Ctrl* + *Z* the user could press and release the *Shift* key to latch the *Shift* modifier, then press and release the *Ctrl* key to latch the *Control* modifier — the *Ctrl* key is a modifier key, so pressing it does not unlatch the *Shift* modifier, but leaves both the *Shift* and *Control* modifiers latched, instead. When the user presses the *Z* key, it will be as though the user pressed *Shift + Ctrl + Z* simultaneously. The *Z* key is not a modifier key, so the *Shift* and *Control* modifiers are unlatched after the event is generated.

A locked a modifier remains in effect until the user unlocks it. For example, to enter the sequence ("XKB") on a PC-style keyboard with a typical US/ASCII layout, the user could press and release the *Shift* key twice to lock the *Shift* modifier. Then, when the user presses the *9* , *'* , *x* , *k* , *b* , *'* , and *0* keys in sequence, it will generate ("XKB"). To unlock the *Shift* modifier, the user can press and release the *Shift* key.

Two option flags modify the behavior of the *StickyKeys* control:

- If the *XkbAX_TwoKeys* flag is set, XKB automatically turns *StickyKeys* off if the user presses two or more keys at once. This serves to automatically disable StickyKeys when a user who does not require sticky keys is using the keyboard.
- The *XkbAX_LatchToLock* controls the locking behavior of *StickyKeys* ; the *StickyKeys* control only locks modifiers as described above if the *XkbAX_LatchToLock* flag is set.

The MouseKeys Control

The *MouseKeys* control lets a user control all the mouse functions from the keyboard. When *MouseKeys* are enabled, all keys with *MouseKeys* actions bound to them generate core pointer events instead of normal key press and release events.

The *MouseKeys* control has a single parameter, the *mouse keys default button* , which specifies the core pointer button to be used by mouse keys actions that do not explicitly specify a button.

The MouseKeysAccel Control

If the *MouseKeysAccel* control is enabled, the effect of a pointer motion action changes as a key is held down. The *mouse keys delay* specifies the amount of time between the initial key press and the first repeated motion event. The *mouse keys interval* specifies the amount of time between repeated mouse keys events. The *steps to maximum acceleration* field specifies the total number of events before the key is travelling at maximum speed. The *maximum acceleration* field specifies the maximum acceleration. The *curve* parameter controls the ramp used to reach maximum acceleration.

When *MouseKeys* are active and a *SA_MovePtr* key action (see Key Actions) is activated, a pointer motion event is generated immediately. If *MouseKeysAccel* is enabled and if acceleration is enabled for the key in question, a second event is generated after *mouse keys delay* milliseconds, and additional events are generated every *mouse keys interval* milliseconds for as long as the key is held down.

Relative Pointer Motion

If the *SA_MovePtr* action specifies relative motion, events are generated as follows: The initial event always moves the cursor the distance specified in the action; after *steps to maximum acceleration* events have been generated, all subsequent events move the pointer the distance specified in the action times the *maximum acceleration.* Events after the first but before maximum acceleration has been achieved are accelerated according to the formula:

$$d(step) = action\_delta \times \left(\frac{max\_accel}{steps\_to\_max^{\,curveFactor}} \right) \times step^{\,curveFactor}$$

Where *action_delta* is the offset specified by the mouse keys action, *max_accel* and *steps_to_max* are parameters to the *MouseKeysAccel* ctrl, and the curveFactor is computed using the *MouseKeysAccel* *curve* parameter as follows:

$$curveFactor(curve) = 1 + \frac{curve}{1000}$$

With the result that a *curve* of *0* causes the distance moved to increase linearly from *action_delta* to

$$(max\_accel \times action\_delta)$$

, and the minimum legal *curve* of - *1000* causes all events after the first move at *max_accel* . A negative *curve* causes an initial sharp increase in acceleration which tapers off, while a positive curve yields a slower initial increase in acceleration followed by a sharp increase as the number of pointer events generated by the action approaches *steps_to_max* .

Absolute Pointer Motion

If an *SA_MovePtr* action specifies an absolute position for one of the coordinates but still allows acceleration, all repeated events contain any absolute coordinates specified in the action.

The AccessXKeys Control

If *AccessXKeys* is enabled many controls can also be turned on or off from the keyboard by entering the following standard key sequences:

- Holding down a shift key by itself for eight seconds toggles the *SlowKeys* control.
- Pressing and releasing a shift key five times in a row without any intervening key events and with less than 30 seconds delay between consecutive presses toggles the state of the *StickyKeys* control.
- Simultaneously operating two or more modifier keys deactivates the *StickyKeys* control.

Some of these key sequences optionally generate audible feedback of the change in state, as described in The AccessXFeedback Control, or cause *XkbAccessXNotify* events as described in Events.

The AccessXTimeout Control

In environments where computers are shared, features such as *SlowKeys* present a problem: if *SlowKeys* is on, the keyboard can appear to be unresponsive because keys have no effect unless they are held for a certain period of time. To help address this problem, XKB provides an *AccessXTimeout* control to automatically change the value of any global controls or AccessX options if the keyboard is idle for a specified period of time.

The AccessXTimeout control has a number of parameters which affect the duration of the timeout and the features changed when the timeout expires.

The *AccessX Timeout* field specifies the number of seconds the keyboard must be idle before the global controls and AccessX options are modified. The *AccessX Options Mask* field specifies which values in the *AccessX Options* field are to be changed, and the *AccessX Options Values* field specifies the new values for those options. The *AccessX Controls Mask* field specifies which controls are to be changed in the global set of *enabled controls* , and the *AccessX Controls Values* field specifies the new values for those controls.

The AccessXFeedback Control

If *AccessXFeedback* is enabled, special beep-codes indicate changes in keyboard controls (or some key events when *SlowKeys* or *StickyKeys* are active). Many beep codes sound as multiple tones, but XKB reports a single *XkbBellNotify* event for the entire sequence of tones.

All feedback tones are governed by the *AudibleBell* control. Individual feedback tones can be explicitly enabled or disabled using the *accessX options mask* or set to deactivate after an idle period using the *accessX timeout options mask* . XKB defines the following feedback tones:

Feedback Name	**Bell Name**	**Default Sound**	**Indicates**
FeatureFB	AX_FeatureOn	rising tone	Keyboard control enabled
	AX_FeatureOff	falling tone	Keyboard control disabled
	AX_FeatureChange	two tones	Several controls changed state
IndicatorFB	AX_IndicatorOn	high tone	Indicator Lit
	AX_IndicatorOff	low tone	Indicator Extinguished
	AX_IndicatorChange	two high tones	Several indicators changed state
SlowWarnFB	AX_SlowKeysWarning	three high tones	Shift key held for four seconds
SKPressFB	AX_SlowKeyPress	single tone	Key press while *SlowKeys* are on
SKReleaseFB	AX_SlowKeyRelease	single tone	Key release while *SlowKeys* are on
SKAcceptFB	AX_SlowKeyAccept	single tone	Key event accepted by *SlowKeys*
SKRejectFB	AX_SlowKeyReject	low tone	Key event rejected by *SlowKeys*
StickyKeysFB	AX_StickyLatch	low tone then high tone	Modifier latched by *StickyKeys*
	AX_StickyLock	high tone	Modifier locked by *StickyKeys*
	AX_StickyUnlock	low tone	Modifier unlocked by *StickyKeys*
BKRejectFB	AX_BounceKeysReject	low tone	Key event rejected by *BounceKeys*

Implementations that cannot generate continuous tones may generate multiple beeps instead of falling and rising tones; for example, they can generate a high-pitched beep followed by a low-pitched beep instead of a continuous falling tone.

If the physical keyboard bell is not very capable, attempts to simulate a continuous tone with multiple bells can sound horrible. Set the *DumbBellFB* AccessX option to inform the server that the keyboard bell is not very capable and that XKB should use only simple bell combinations. Keyboard capabilities vary wildly, so the sounds generated for the individual bells when the *DumbBellFB* option is set are implementation specific.

The Overlay1 and Overlay2 Controls

A keyboard overlay allows some subset of the keyboard to report alternate keycodes when the overlay is enabled. For example a keyboard overlay can be used to simulate a numeric or editing keypad on keyboard that does not actually have one by generating alternate of keycodes for some keys when the overlay is enabled. This technique is very common on portable computers and embedded systems with small keyboards.

XKB includes direct support for two keyboard overlays, using the *Overlay1* and *Overlay2* controls. When *Overlay1* is enabled, all of the keys that are members of the first keyboard overlay generate an alternate keycode. When *Overlay2* is enabled, all of the keys that are members of the second keyboard overlay generate an alternate keycode.

To specify the overlay to which a key belongs and the alternate keycode it should generate when that overlay is enabled, assign it either the *KB_Overlay1* or *KB_Overlay2* key behaviors, as described in Key Behavior.

"Boolean" Controls and The EnabledControls Control

All of the controls described above, along with the *AudibleBell* control (described in Disabling Server Generated Bells) and the *IgnoreGroupLock* control (described in Server Internal Modifiers and Ignore Locks Behavior) comprise the *boolean controls* . In addition to any parameters listed in the descriptions of the individual controls, the boolean controls can be individually enabled or disabled by changing the value of the *EnabledControls* control.

The following *non-boolean* controls are always active and cannot be changed using the *EnabledControls* control or specified in any context that accepts only boolean controls: *GroupsWrap* (Computing Effective Modifier and Group), *EnabledControls* , *InternalMods* (Server Internal Modifiers and Ignore Locks Behavior), and *IgnoreLockMods* (Server Internal Modifiers and Ignore Locks Behavior) and *PerKeyRepeat* (The RepeatKeys Control)

Automatic Reset of Boolean Controls

The *auto-reset controls* are a per-client value which consist of two masks that can contain any of the boolean controls (see "Boolean" Controls and The EnabledControls Control). Whenever the client exits for any reason, any boolean controls specified in the *auto-reset mask* are set to the corresponding value from the *auto-reset values* mask. This makes it possible for clients to "clean up after themselves" automatically, even if abnormally terminated.

For example, a client that replace the keyboard bell with some other audible cue might want to turn off the *AudibleBell* control (Disabling Server Generated Bells) to prevent the server from also generating a sound and thus avoid cacophony. If the client were to exit without resetting the *AudibleBell* control, the user would be left without any feedback at all. Setting *AudibleBell* in both the auto-reset mask and auto-reset values guarantees that the audible bell will be turned back on when the client exits.

Chapter 5. Key Event Processing Overview

There are three steps to processing each key event in the X server, and at least three in the client. This section describes each of these steps briefly; the following sections describe each step in more detail.

1. First, the server applies global keyboard controls to determine whether the key event should be processed immediately, deferred, or ignored. For example, the *SlowKeys* control can cause a key event to be deferred until the slow keys delay has elapsed while the *RepeatKeys* control can cause multiple X events from a single physical key press if the key is held down for an extended period. The global keyboard controls affect all of the keys on the keyboard and are described in Global Keyboard Controls.

2. Next, the server applies per-key behavior. Per key-behavior can be used to simulate or indicate some special kinds of key behavior. For example, keyboard overlays, in which a key generates an alternate keycode under certain circumstances, can be implemented using per-key behavior. Every key has a single behavior, so the effect of key behavior does not depend on keyboard modifier or group state, though it might depend on global keyboard controls. Per-key behaviors are described in detail in Key Behavior.

3. Finally, the server applies key actions. Logically, every keysym on the keyboard has some action associated with it. The key action tells the server what to do when an event which yields the corresponding keysym is generated. Key actions might change or suppress the event, generate some other event, or change some aspect of the server. Key actions are described in Key Actions.

If the global controls, per-key behavior and key action combine to cause a key event, the client which receives the event processes it in several steps.

1. First the client extracts the effective keyboard group and a set of modifiers from the state field of the event. See Computing A State Field from an XKB State for details.

2. Using the modifiers and effective keyboard group, the client selects a symbol from the list of keysyms bound to the key. Determining the KeySym Associated with a Key Event discusses symbol selection.

3. If necessary, the client transforms the symbol and resulting string using any modifiers that are "left over" from the process of looking up a symbol. For example, if the *Lock* modifier is left over, the resulting keysym is capitalized according to the capitalization rules specified by the system. See Transforming the KeySym Associated with a Key Event for a more detailed discussion of the transformations defined by XKB.

4. Finally, the client uses the keysym and remaining modifiers in an application-specific way. For example, applications based on the X toolkit might apply translations based on the symbol and modifiers reported by the first three steps.

Chapter 6. Key Event Processing in the Server

This section describes the steps involved in processing a key event within the server when XKB is present. Key events can be generated due to keyboard activity and passed to XKB by the DDX layer, or they can be synthesized by another extension, such as XTEST.

Applying Global Controls

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

- If a key is pressed while the *BounceKeys* control is enabled, the extension generates the 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 *SlowKeys* 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 key release deactivates this timer.

  If the slow keys timer expires, the server generates a key press for the corresponding key, sends an *XkbAccessXNotify* and deactivates the timer.
- The extension processes key press events normally whether or not the *RepeatKeys* control is active, but if *RepeatKeys* are enabled and per-key autorepeat is enabled for the event key, the extension processes key press events normally, but it also initiates an *autorepeat timer* with an interval specified by the autorepeat delay. 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.

Key events are processed by each global control in turn: if the *BounceKeys* control accepts a key event, *SlowKeys* considers it. Once *SlowKeys* allows or synthesizes an event, the *RepeatKeys* control acts on it.

Key Behavior

Once an event is accepted by all of the controls or generated by a timer, the server checks the per-key behavior of the corresponding key. This extension currently defines the following key behaviors:

Behavior	**Effect**
KB_Default	Press and release events are processed normally.
KB_Lock	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.
KB_RadioGroup flags: CARD8 index: CARD8	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 processing of the corresponding key release depends on the value of the *RGAllowNone* bit in *flags* . If it is set, the key release is processed normally; otherwise the key release is also ignored. All other key release events are ignored.
KB_Overlay1 key: KEYCODE	If the *Overlay1* control is enabled, events from this key are reported as if they came from the key specified in *key* . Otherwise, press and release events are processed normally.
KB_Overlay2 key: KEYCODE	If the *Overlay2* control is enabled, events from this key are reported as if they came from the key specified in *key* . Otherwise, press and release events are processed normally.

The X server uses key behavior to determine whether to process or filter out any given key event; key behavior is independent of keyboard modifier or group state (each key has exactly one behavior.

Key behaviors can be used to simulate any of these types of keys or to indicate an unmodifiable physical, electrical or software driver characteristic of a key. An optional *permanent* flag can modify any of the supported behaviors and indicates that behavior describes an unalterable physical, electrical or software aspect of the keyboard. Permanent behaviors cannot be changed or set by the *XkbSetMap* request. The *permanent* flag indicates a characteristic of the underlying system that XKB cannot affect, so XKB treats all permanent behaviors as if they were *KB_Default* and does not filter key events described in the table above.

Key Actions

Once the server has applied the global controls and per-key behavior and has decided to process a key event, it applies *key actions* to determine the effects of the key on the internal state of the server. A key action 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 most aspects of keyboard behavior

- terminate or suspend the server
- send a message to interested clients
- simulate events on other keys

Each key has an optional list of actions. If present, this list parallels the list of symbols associated with the key (i.e. it has one action per symbol associated with the key). For key press events, the server looks up the action to be applied from this list using the key symbol mapping associated with the event key, just as a client looks up symbols as described in Determining the KeySym Associated with a Key Event; if the event key does not have any actions, the server uses the *SA_NoAction* event for that key regardless of modifier or group state.

Key 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 key press event determines the further actions, if any, that are applied to the corresponding release event or to events that occur while the key is held down. Clients can change the actions associated with a key while the key is down without changing the action applied next time the key is released; subsequent press-release pairs will use the newly bound key action.

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 or delay between the activation of one and the other.

Most actions which affect keyboard modifier state accept a modifier definition (see Virtual Modifiers) named *mods* and a boolean flag name *useModMap* among their arguments. These two fields combine to specify the modifiers affected by the action as follows: If *useModMap* is *True* , the action sets any modifiers bound by the modifier mapping to the key that initiated the action; otherwise, the action sets the modifiers specified by *mods* . For brevity in the text of the following definitions, we refer to this combination of *useModMap* and *mods* as the "action modifiers."

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 mods: MOD_DEF useModMap: BOOL clearLocks: BOOL	• Key press adds any action modifiers to the keyboard’s base modifiers. • Key release clears any action modifiers in the keyboard’s base modifiers, provided that no other key which affects the same modifiers is logically down. • If no keys were operated simultaneously with this key and *clearLocks* is set, release unlocks any action modifiers.
SA_LatchMods mods: MOD_DEF useModMap: BOOL	• Key press and 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:

Action	Effect
clearLocks: BOOL latchToLock: BOOL	• Modifiers that were unlocked due to *clearLocks* have no further effect. • If *latchToLock* is set, key release locks and then unlatches any remaining action modifiers that are already latched. • Finally, key release latches any action modifiers that were not used by the *clearLocks* or *latchToLock* flags.
SA_LockMods mods: MOD_DEF useModMap: BOOL noLock: BOOL noUnlock: BOOL	• Key press sets the base and possibly the locked state of any action modifiers. If *noLock* is *True* , only the base state is changed. • For key release events, clears any action modifiers in the keyboard's base modifiers, provided that no other key which affects the same modifiers is down. If *noUnlock* is *False* and any of the action modifiers were locked before the corresponding key press occurred, key release unlocks them.
SA_SetGroup group: INT8 groupAbsolute: BOOL clearLocks: BOOL	• If *groupAbsolute* 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 resulting effective keyboard group is brought back into range depending on the value of the *GroupsWrap* control 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 press. • If no keys were operated simultaneously with this key and *clearLocks* is set, key release also sets the locked keyboard group to *Group1* .
SA_LatchGroup group: INT8 groupAbsolute: BOOL clearLocks: BOOL latchToLock: BOOL	• 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 the *clearLocks* flag 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. The locked and effective keyboard group are brought back into range according to the value of the global *GroupsWrap* control for the keyboard. • Otherwise, key release adds the key press delta to the latched keyboard group.
SA_LockGroup group: INT8	• If *groupAbsolute* is set, key press sets the locked keyboard group to *group* . Otherwise, key press adds *group* to the locked keyboard group. In either case, the resulting locked and effective group is brought back into range de-

Action	**Effect**
groupAbsolute: BOOL	pending on the value of the *GroupsWrap* control for the keyboard. • Key release has no effect.
SA_MovePtr x, y: INT16 noAccel: BOOL absoluteX: BOOL absoluteY: BOOL	• If *MouseKeys* are not enabled, this action behaves like *SA_NoAction*, otherwise this action cancels any pending repeat key timers for this key and has the following additional effects. • Key press generates a core pointer *MotionNotify* event instead of the usual *KeyPress*. If *absoluteX* is *True*, *x* specifies the new pointer X coordinate, otherwise *x* is added to the current pointer X coordinate; *absoluteY* and *y* specify the new Y coordinate in the same way. • If *noAccel* is *False*, and the *MouseKeysAccel* keyboard 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 as described in The MouseKeysAccel Control. • Key release disables the mouse keys timer (if it was initiated by the corresponding key press) but has no other effect and is ignored (does not generate an event of any type).
SA_PtrBtn button: CARD8 count: CARD8 useDfltBtn: BOOL	• If *MouseKeys* are not enabled, this action behaves like *SA_NoAction*. • If *useDfltBtn* is set, the event is generated for the current default core button. Otherwise, the event is generated for the button specified by *button*. • 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 causes one or more core pointer button events instead of the usual key press. If *count* is *0*, key press generates a single *ButtonPress* event; if *count* is greater than *0*, key press generates *count* pairs of *ButtonPress* and *ButtonRelease* events. • If *count* is *0*, key release generates a core pointer *ButtonRelease* which matches the event generated by the corresponding key press; if count is non-zero, key release does not cause a *ButtonRelease* event. Key release never causes a key release event.
SA_LockPtrBtn button: BUTTON noLock: BOOL	• If *MouseKeys* are not enabled, this action behaves like *SA_NoAction*. • Otherwise, if the button specified by *useDfltBtn* and *button* is not locked, key press causes a *ButtonPress* instead of a key press and locks the button. If the button is already

Action	Effect
noUnlock: BOOL useDfltBtn: BOOL	locked or if *noLock* is *True* , key press is ignored and has no effect. • If the corresponding key press was ignored, and if *noUnlock* is *False* , key release generates a *ButtonRelease* event instead of a key release event and unlocks the specified button. If the corresponding key press locked a button, key release is ignored and has no effect.
SA_SetPtrDflt affect: CARD8 value: CARD8 dfltBtnAbs: BOOL	• If *MouseKeys* are not enabled, this action behaves like *SA_NoAction* . • Otherwise, both key press and key release are ignored, but key press changes the pointer value specified by *affect* to *value* , as follows: • If *which* is *SA_AffectDfltBtn* , *value* and *dfltBtnAbs* specify the default pointer button used by the various pointer actions as follow: If *dfltBtnAbs* is True, value specifies the button to be used, otherwise, *value* specifies the amount to be added to the current default button. In either case, illegal button choices are wrapped back into range.
SA_ISOLock dfltIsGroup: *False* mods: MOD_DEF useModMap: BOOL noLock: BOOL noUnlock: BOOL noAffectMods: BOOL noAffectGrp: BOOL noAffectPtr: BOOL noAffectCtrls: BOOL or dfltIsGroup: *True* group: INT8 groupAbsolute: BOOL noAffectMods: BOOL	• If *dfltIsGroup* is *True* , key press sets the base group specified by *groupAbsolute* and *group* . Otherwise, key press sets the action modifiers in the keyboard’s base modifiers. • 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 *noAffectGrp* is *False* , any *SA_SetGroup* or *SA_LatchGroup* actions that occur simultaneously with this action are treated as *SA_LockGroup* actions instead. • If *noAffectPtr* is *False* , *SA_PtrBtn* actions that occur simultaneously with the *SA_ISOLock* action are treated as *SA_LockPtrBtn* actions instead. • If *noAffectCtrls* is *False* , any *SA_SetControls* actions that occur simultaneously 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.

Action	**Effect**
noAffectGrp: BOOL noAffectPtr: BOOL noAffectCtrls: BOOL	
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 num: INT8 switchApp: BOOL screenAbs: BOOL	• If the server supports this action and multiple screens or displays (either virtual or real), this action changes to the active screen indicated by *num* and *screenAbs* . If *screenAbs* is *True* , num specifies the index of the new screen; otherwise, num specifies an offset from the current screen to the new screen. • If *switchApp* is *False* , it should switch to another screen on the same server. Otherwise it should switch to another X server or application which shares the same physical display. • This action is optional; servers are free to ignore the action or any of its flags if they do not support the requested behavior. If the action is ignored, it behaves like *SA_NoAction* , otherwise neither key press nor release generate an event.
SA_SetControls controls: KB_BOOLCTRLMASK	• Key press enables any boolean 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 *XkbControlsNotify* events.
SA_LockControls controls: KB_BOOLCTRLMASK noLock: BOOL noUnlock: BOOL	• If *noLock* is *False* , key press locks and enables any controls that are specified in *controls* and not already locked at the time of the key press. If *noUnlock* is *False* , key release unlocks and disables any controls that are specified in *controls* and were not enabled at the time of the corresponding key press.
SA_ActionMessage : pressMsg: BOOL releaseMsg: BOOL genEvent: BOOL message: STRING	• if *pressMsg* is *True* , key press generates an *XkbActionMessage* event which reports the keycode, event type and the contents of *message* . • If *releaseMsg* is *True* , key release generates an *XkbActionMessage* event which reports the keycode, event type and contents of *message* . • If *genEvent* is *True* , both press and release generate key press and key release events, regardless of whether they also cause an *XkbActionMessage* .

Action	**Effect**
SA_RedirectKey newKey: KEYCODE modsMask: KEYMASK mods: KEYMASK vmodsMask: CARD16 vmods: CARD16	• Key press causes a key press event for the key specified by *newKey* instead of for the actual key. The state reported in this event reports of the current effective modifiers changed as follow: Any real modifiers specified in *modsMask* are set to corresponding values from *mods* . Any real modifiers bound to the virtual modifiers specified in *vmodsMask* are either set or cleared, depending on the corresponding value in *vmods* . If the real and virtual modifier definitions specify conflicting values for a single modifier, the real modifier definition has priority. • Key release causes a key release event for the key specified by *newKey* ; the state field for this event consists of the effective keyboard modifiers at the time of the release, changed as described above. • The *SA_RedirectKey* action normally redirects to another key on the same device as the key or button which caused the event, unless that device does not belong to the input extension KEYCLASS, in which case this action causes an event on the core keyboard device.
SA_DeviceBtn count: CARD8 button: BUTTON device: CARD8	• The *device* field specifies the ID of an extension device; the *button* field specifies the index of a button on that device. If the button specified by this action is logically down, the key press and corresponding release are ignored and have no effect. If the device or button specified by this action are illegal, this action behaves like *SA_NoAction* . • Otherwise, key press causes one or more input extension device button events instead of the usual key press event. If *count* is *0* , key press generates a single *DeviceButtonPress* event; if *count* is greater than *0* , key press generates *count* pairs of *DeviceButtonPress* and *DeviceButtonRelease* events. • If *count* is *0* , key release generates an input extension *DeviceButtonRelease* which matches the event generated by the corresponding key press; if count is non-zero, key release does not cause a *DeviceButtonRelease* event. Key release never causes a key release event.
SA_LockDeviceBtn button: BUTTON device: CARD8 noLock: BOOL noUnlock: BOOL	• The *device* field specifies the ID of an extension device; the *button* field specifies the index of a button on that device. If the device or button specified by this action are illegal, it behaves like *SA_NoAction* . • Otherwise, if the specified button is not locked and if *noLock* is *False* , key press causes an input extension *DeviceButtonPress* event instead of a key press event and locks the button. If the button is already locked or if *noLock* is *True* , key press is ignored and has no effect.

Action	**Effect**
	• If the corresponding key press was ignored, and if *noUnlock* is *False* , key release generates an input extension *DeviceButtonRelease* event instead of a core protocol or input extension key release event and unlocks the specified button. If the corresponding key press locked a button, key release is ignored and has no effect.
SA_DeviceValuator *device* : CARD8 *val1What* : SA_DVOP *val1* : CARD8 *val1Value* : INT8 *val1Scale* : 0...7 *val2What* : BOOL *val2* : CARD8 *val2Value* : INT8 *val2Scale* : 0...7	• The *device* field specifies the ID of an extension device; *val1* and *val2* specify valuators on that device. If *device* is illegal or if neither *val1* nor *val2* specifies a legal valuator, this action behaves like *SA_NoAction* . • If *valn* specifies a legal valuator and *valnWhat* is not *SA_IgnoreVal* , the specified value is adjusted as specified by *valnWhat* : • If *valnWhat* is *SA_SetValMin* , *valn* is set to its minimum legal value. • If *valnWhat* is *SA_SetValCenter* , *valn* is centered (to (max-min)/2). • If *valnWhat* is *SA_SetValMax* , *valn* is set to its maximum legal value. • if *valnWhat* is *SA_SetValRelative* , $valnValue \times 2^{valnScale}$ is added to *valn* . • if *valnWhat* is *SA_SetValAbsolute* , *valn* is set to $valnValue \times 2^{valnScale}$. • Illegal values for *SA_SetValRelative* or *SA_SetValAbsolute* are clamped into range.

If *StickyKeys* are enabled, all *SA_SetMods* and *SA_SetGroup* actions act like *SA_LatchMods* and *SA_LatchGroup* respectively. If the *LatchToLock* AccessX option is set, either action behaves as if both the *SA_ClearLocks* and *SA_LatchToLock* flags are set.

Actions which cause an event from another key or from a button on another device immediately generate the specified event. These actions do not consider the behavior or actions (if any) that are bound to the key or button to which the event is redirected.

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 an *XkbUseExtension* request contain a compatibility state in place of the actual XKB keyboard state. See Effects of XKB on Core Protocol Events for a description of this compatibility mapping.

Delivering a Key or Button Event to a Client

The window and client that receive core protocol and input extension key or button events are determined using the focus policy, window hierarchy and passive grabs as specified by the core protocol and the input extension, with the following changes:

- A passive grab triggers if the modifier state specified in the grab matches the grab compatibility state (described in Compatibility Components of Keyboard State). Clients can choose to use the XKB grab state instead by setting the *GrabsUseXKBState* per-client flag. This flag affects all passive grabs that are requested by the client which sets it but does not affect passive grabs that are set by any other client.

- The state field of events which trigger a passive grab reports the XKB or compatibility grab state in effect at the time the grab is triggered; the state field of the corresponding release event reports the corresponding grab state in effect when the key or button is released.

- If the *LookupStateWhenGrabbed* per-client flag is set, all key or button events that occur while a keyboard or pointer grab is active contain the XKB or compatibility lookup state, depending on the value of the *GrabsUseXKBState* per-client flag. If *LookupStateWhenGrabbed* is not set, they include the XKB or compatibility grab state, instead.

- Otherwise, the state field of events that do not trigger a passive grab report is derived from the XKB effective modifiers and group, as described in Computing A State Field from an XKB State.

- If a key release event is the result of an autorepeating key that is being held down, and the client to which the event is reported has requested detectable autorepeat (see Detectable Autorepeat), the event is not delivered to the client.

The following section explains the intent of the XKB interactions with core protocol grabs and the reason that the per-client flags are needed.

XKB Interactions With Core Protocol Grabs

XKB provides the separate lookup and grab states to help work around some difficulties with the way the core protocol specifies passive grabs. Unfortunately, many clients work around those problems differently, and the way that XKB handles grabs and reports keyboard state can sometimes interact with those client workarounds in unexpected and unpleasant ways.

To provide more reasonable behavior for clients that are aware of XKB without causing problems for clients that are unaware of XKB, this extension provides two per-client flags that specify the way that XKB and the core protocol should interact.

- The largest problems arise from the fact that an XKB state field encodes an explicit keyboard group in bits 13-14 (as described in Computing A State Field from an XKB State), while pre-XKB clients use one of the eight keyboard modifiers to select an alternate keyboard group. To make existing clients behave reasonably, XKB normally uses the compatibility grab state instead of the XKB grab state to determine whether or not a passive grab is triggered. XKB-aware clients can set the *GrabsUseXKBState* per-client flag to indicate that they are specifying passive grabs using an XKB state.

- Some toolkits start an active grab when a passive grab is triggered, in order to have more control over the conditions under which the grab is terminated. Unfortunately, the fact that XKB reports a different state in events that trigger or terminate grabs means that this grab simulation can fail to terminate the grab under some conditions. To work around this problem, XKB normally reports the grab state in all events whenever a grab is active. Clients which do not use active grabs like this can set the *LookupStateWhenGrabbed* per-client flag in order to receive the same state component whether or not a grab is active.

  The *GrabsUseXKBState* per-client flag also applies to the state of events sent while a grab is active. If it is set, events during a grab contain the XKB lookup or grab state; by default, events during a grab contain the compatibility lookup or grab state.

The state used to trigger a passive grab is controlled by the setting of the *GrabsUseXKBState* per-client flag at the time the grab is registered. Changing this flag does not affect existing passive grabs.

Chapter 7. Key Event Processing in the Client

The XKB *client map* for a keyboard is the collection of information a client needs to interpret key events that come from that keyboard. It contains a global list of *key types* , described in Key Types, and an array of *key symbol map* s, each of which describes the symbols bound to one particular key and the rules to be used to interpret those symbols.

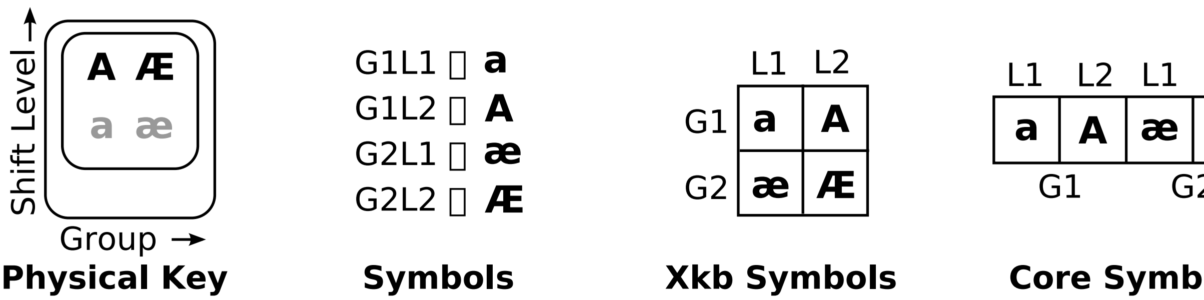
Notation and Terminology

XKB associates a two-dimensional array of symbols with each key. Symbols are addressed by keyboard group (see Keyboard State) and shift level, where level is defined as in the ISO9995 standard:

Level One of several states (normally 2 or 3) which govern which graphic character is produced when a graphic key is actuated. In certain cases the level may also affect function keys.

Note that shift level is derived from the modifier state, but not necessarily in the same way for all keys. For example, the *Shift* modifier selects shift level 2 on most keys, but for keypad keys the modifier bound to *Num_Lock* (i.e. the *NumLock* virtual modifier) also selects shift level 2.gray symbols on a key

We use the notation G *n* L *n* to specify the position of a symbol on a key or in memory:



The gray characters indicate symbols that are implied or expected but are not actually engraved on the key.

Note

Unfortunately, the "natural" orientation of symbols on a key and the natural orientation in memory are reversed from one another, so keyboard group refers to a column on the key and a row in memory. There's no real help for it, but we try to minimize confusion by using "group" and "level" (or "shift level") to refer to symbols regardless of context.

Determining the KeySym Associated with a Key Event

To look up the symbol associated with an XKB key event, we need to know the group and shift level that correspond to the event.

Group is reported in bits 13-14 of the state field of the key event, as described in Computing A State Field from an XKB State. The keyboard group reported in the event might be out-of-range for any particular key because the number of groups can vary from key to key. The XKB description of each key contains a *group info* field which is interpreted identically to the global groups wrap control (see Computing Effective Modifier and Group) and which specifies the interpretation of groups that are out-of-range for that key.

Once we have determined the group to be used for the event, we have to determine the shift level. The description of a key includes a *key type* for each group of symbols bound to the key. Given the modifiers from the key event, this key type yields a shift level and a set of "leftover" modifiers, as described in Key Types below.

Finally, we can use the effective group and the shift level returned by the type of that group to look up a symbol in a two-dimensional array of symbols associated with the key.

Key Types

Each entry of a key type's *map* field specifies the shift level that corresponds to some XKB modifier definition; any combination of modifiers that is not explicitly listed somewhere in the map yields shift level one. Map entries which specify unbound virtual modifiers (see Inactive Modifier Definitions) are not considered; each entry contains an automatically-updated *active* field which indicates whether or not it should be used.

Each key type includes a few fields that are derived from the contents of the map and which report some commonly used values so they don't have to be constantly recalculated. The *numLevels* field contains the highest shift level reported by any of its map entries; XKB uses *numLevels* to insure that the array of symbols bound to a key is large enough (the number of levels reported by a key type is also referred to as its width). The *modifiers* field reports all real modifiers considered by any of the map entries for the type. Both *modifiers* and *numLevels* are updated automatically by XKB and neither can be changed explicitly.

Any modifiers specified in *modifiers* are normally *consumed* (see Transforming the KeySym Associated with a Key Event), which means that they are not considered during any of the later stages of event processing. For those rare occasions that a modifier *should* be considered despite having been used to look up a symbol, key types include an optional *preserve* field. If a *preserve* list is present, each entry corresponds to one of the key type's map entries and lists the modifiers that should *not* be consumed if the matching map entry is used to determine shift level.

For example, the following key type implements caps lock as defined by the core protocol (using the second symbol bound to the key):

```
type "ALPHABETIC" {
 modifiers = Shift+Lock;
 map[Shift]= Level2;
 map[Lock]= Level2;
 map[Shift+Lock]= Level2;
};
```

The problem with this kind of definition is that we could assign completely unrelated symbols to the two shift levels, and "Caps Lock" would choose the second symbol. Another definition for alphabetic keys uses system routines to capitalize the keysym:

```
type "ALPHABETIC" {
 modifiers= Shift;
 map[Shift]= Level2;
};
```

When caps lock is applied using this definition, we take the symbol from shift level one and capitalize it using system-specific capitalization rules. If shift and caps lock are both set, we take the symbol from shift level two and try to capitalize it, which usually has no effect.

The following key type implements shift-cancels-caps lock behavior for alphabetic keys:

```
type "ALPHABETIC" {
 modifiers = Shift+Lock;
 map[Shift] = Level2;
 preserve[Lock]= Lock;
};
```

Consider the four possible states that can affect alphabetic keys: no modifiers, shift alone, caps lock alone or shift and caps lock together. The map contains no explicit entry for *None* (no modifiers), so if no modifiers are set, any group with this type returns the first keysym. The map entry for *Shift* reports *Level2* , so any group with this type returns the second symbol when *Shift* is set. There is no map entry for *Lock* alone, but the type specifies that the *Lock* modifier should be preserved in this case, so *Lock* alone returns the first symbol in the group but first applies the capitalization transformation, yielding the capital form of the symbol. In the final case, there is no map entry for *Shift+Lock* , so it returns the first symbol in the group; there is no preserve entry, so the *Lock* modifier is consumed and the symbol is not capitalized.

Key Symbol Map

The *key symbol map* for a key contains all of the information that a client needs to process events generated by that key. Each key symbol mapping reports:

- The number of groups of symbols bound to the key (*numGroups*).
- The treatment of out-of-range groups (*groupInfo*).
- The index of the key type to for each *possible* group (*kt_index[MaxKbdGroups]*).

- The width of the widest type associated with the key (*groupsWidth*).
- The two-dimensional (numGroups × groupsWidth) array of symbols bound to the key.

It is legal for a key to have zero groups, in which case it also has zero symbols and all events from that key yield *NoSymbol* . The array of key types is of fixed width and is large enough to hold key types for the maximum legal number of groups (*MaxKbdGroups* , currently four); if a key has fewer than *MaxKbdGroups* groups, the extra key types are reported but ignored. The *groupsWidth* field cannot be explicitly changed; it is updated automatically whenever the symbols or set of types bound to a key are changed.

If, when looking up a symbol, the effective keyboard group is out-of-range for the key, the *groupInfo* field of the key symbol map specifies the rules for determining the corresponding legal group as follows:

- If the *RedirectIntoRange* flag is set, the two least significant bits of *groupInfo* specify the index of a group to which all illegal groups correspond. If the specified group is also out of range, all illegal groups map to *Group1* .
- If *ClampIntoRange* flag is set, out-of-range groups correspond to the nearest legal group. Effective groups larger than the highest supported group are mapped to the highest supported group; effective groups less than *Group1* are mapped to *Group1* . For example, a key with two groups of symbols uses *Group2* type and symbols if the global effective group is either *Group3* or *Group4* .
- If neither flag is set, group is wrapped into range using integer modulus. For example, a key with two groups of symbols for which groups wrap uses *Group1* symbols if the global effective group is *Group3* or *Group2* symbols if the global effective group is *Group4* .

The client map contains an array of key symbol mappings, with one entry for each key between the minimum and maximum legal keycodes, inclusive. All keycodes which fall in that range have key symbol mappings, whether or not any key actually yields that code.

Transforming the KeySym Associated with a Key Event

Any modifiers that were not used to look up the keysym, or which were explicitly preserved, might indicate further transformations to be performed on the keysym or the character string that is derived from it. For example, If the *Lock* modifier is set, the symbol and corresponding string should be capitalized according to the locale-sensitive capitalization rules specified by the system. If the *Control* modifier is set, the keysym is not affected, but the corresponding character should be converted to a control character as described in Default Symbol Transformations.

This extension specifies the transformations to be applied when the *Control* or *Lock* modifiers are active but were not used to determine the keysym to be used:

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 for a small set of punctuation characters (see Default Symbol Transformations). 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 Default Symbol Transformations).

Interpretation of other modifiers is application dependent.

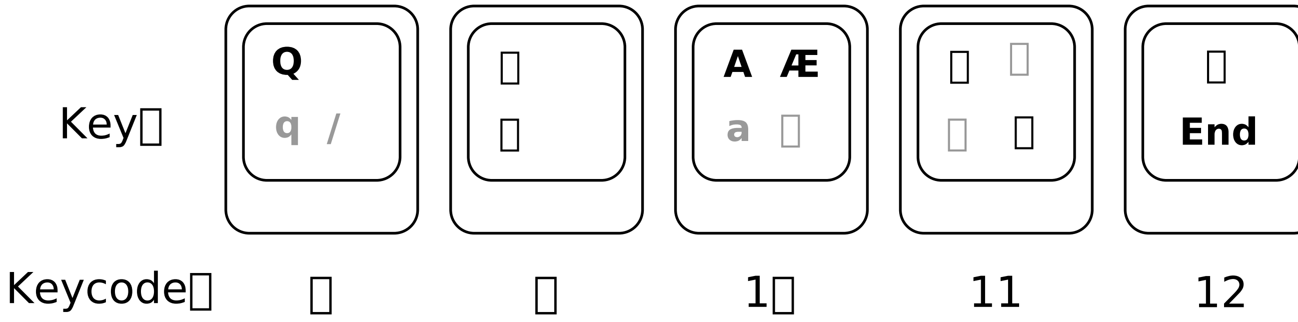
Note

This definition of capitalization is fundamentally different from the core protocol's, which uses the lock modifier to select from the symbols bound to the key. Consider key 9 in the client map example; the core protocol provides no way to generate the capital form of either symbol bound to this key. XKB specifies that we first look up the symbol and then capitalize, so XKB yields the capital form of the two symbols when caps lock is active.

XKB specifies the behavior of *Lock* and *Control* , but interpretation of other modifiers is left to the application.

Client Map Example

Consider a simple, if unlikely, keyboard with the following keys (gray characters indicate symbols that are implied or expected but are not actually engraved on the key):



The core protocol represents this keyboard as a simple array with one row per key and four columns (the widest key, key 10, determines the width of the entire array).

Key	G1L1	G1L2	G2L1	G2L2
8	Q	NoSymbol	at	NoSymbol
9	odiaeresis	egrave	NoSymbol	NoSymbol
10	A	NoSymbol	Æ	NoSymbol
11	ssharp	question	backslash	questiondown
12	KP_End	KP_1	NoSymbol	NoSymbol
13	Num_Lock	NoSymbol	NoSymbol	NoSymbol
14	NoSymbol	NoSymbol	NoSymbol	NoSymbol
15	Return	NoSymbol	NoSymbol	NoSymbol

The row to be used for a given key event is determined by keycode; the column to be used is determined by the symbols bound to the key, the state of the *Shift* and *Lock* Modifiers and the state of the modifiers bound to the *Num_Lock* and *Mode_switch* keys as specified by the core protocol.

The XKB description of this keyboard consists of six key symbol maps, each of which specifies the types and symbols associated with each keyboard group for one key:

Key	Group: Type	L1	L2
8	G1: ALPHABETIC	q	Q
G2: ONE_LEVEL	@	NoSymbol	
9	G1: TWO_LEVEL	odiaeresis	egrave
10	G1: ALPHABETIC	a	A
G2: ALPHABETIC	ae	AE	
11	G1: TWO_LEVEL	ssharp	question
G2: ONE_LEVEL	backslash	questiondown	
12	G1: KEYPAD	KP_End	KP_1
13	G1: ONE_LEVEL	Num_Lock	
14	No Groups		
15	G1: ONE_LEVEL	Return	

The keycode reported in a key event determines the row to be used for that event; the effective keyboard group determines the list of symbols and key type to be used. The key type determines which symbol is chosen from the list.

Determining the KeySym Associated with a Key Event details the procedure to map from a key event to a symbol and/or a string.

Chapter 8. Symbolic Names

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 *keyboard 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 might share a keycodes name but different geometry names.

Note

The keycodes name is intended to be a very general description of the keycodes returned by a keyboard; A single keycodes name 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 geometry to determine which subset of the named keyboard type is in use.

The *symbols* name identifies 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 *physical symbols* name describes the symbols actually engraved on the keyboard, which might be different than the symbols currently being used.

The *types* name provides some information about the set of key types that can be associated with the keyboard keys. The *compat* name provides some information about the rules used to bind actions to keys changed using core protocol requests.

The *compat* , *types* , *keycodes* , *symbols* and *geometry* names typically correspond to the keyboard components from which the current keyboard description was assembled. These components are stored individually in the server's database of keyboard components, described in The Server Database of Keyboard Components, and can be combined to assemble a complete keyboard description.

Each key has a four-byte symbolic name. The key name links keys with similar functions or in similar positions on keyboards that report different scan codes. *Key aliases* allow the keyboard layout designer to assign multiple names to a single key, to make it easier to refer to keys using either their position *or* their "function."

For example, consider the common keyboard customizations:

- Set the "key to the left of the letter a" to be a control key.
- Change the "caps lock" key, wherever it might be, to a control key.

If we specify key names by position, the first customization is simple but the second is impossible; if we specify key names by function, the second customization is simple but the first is impossible. Using key aliases, we can specify both function and position for "troublesome" keys, and both customizations are straightforward.

Key aliases can be specified both in the symbolic names component and in the keyboard geometry (see Keyboard Geometry). Both sets of aliases are always valid, but key alias definitions in the keyboard geometry have priority; if both symbolic names and geometry include aliases, applications should consider the definitions from the geometry before considering the definitions from the symbolic names section.

XKB provides symbolic names for each of the four keyboard groups, sixteen virtual modifiers, thirty-two keyboard indicators, and up to *MaxRadioGroups* (32) radio groups.

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

Chapter 9. Keyboard Indicators

Although the core X protocol supports thirty-two LEDs on a keyboard, it does not provide any way to link the state of the LEDs and the logical state of the keyboard. For example, most keyboards have a "Caps Lock" LED, but X does not provide any standard way to make the LED automatically follow the logical state of the modifier bound to the *Caps Lock* key.

The core protocol also gives no way to determine which bits in the *led_mask* field of the keyboard state map to the particular LEDs on the keyboard. For example, X does not provide a method for a client to determine which bit to set in the *led_mask* to turn on the "Scroll Lock" LED, or even if the keyboard has a "Scroll Lock" LED.

Most X servers implement some kind of automatic behavior for one or more of the keyboard LEDs, but the details of that automatic behavior are implementation-specific and can be difficult or impossible to control.

XKB provides indicator names and programmable indicators to help solve these problems. Using XKB, clients can determine the names of the various indicators, determine and control the way that the individual indicators should be updated to reflect keyboard changes, and determine which of the 32 keyboard indicators reported by the protocol are actually present on the keyboard. Clients may also request immediate notification of changes to the state of any subset of the keyboard indicators, which makes it straightforward to provide an on-screen "virtual" LED panel.

Global Information About Indicators

XKB provides only two pieces of information about the indicators as a group.

The *physical indicators* mask reports which of the 32 logical keyboard indicators supported by the core protocol and XKB corresponds to some actual indicator on the keyboard itself. Because the physical indicators mask describes a physical characteristic of the keyboard, it cannot be directly changed under program control. It is possible, however, for the set of physical indicators to be change if a new keyboard is attached or if a completely new keyboard description is loaded by the *XkbGetKeyboardByName* request (see Using the Server's Database of Keyboard Components).

The *indicator state* mask reports the current state of the 32 logical keyboard indicators. This field and the core protocol indicator state (as reported by the *led-mask* field of the core protocol *GetKeyboardControl* request) are always identical.

Per-Indicator Information

Each of the thirty-two keyboard indicators has a symbolic name, of type ATOM. The *XkbGetNames* request reports the symbolic names for all keyboard components, including the indicators. Use the *XkbSetNames* request to change symbolic names. Both requests are described in Querying and Changing Symbolic Names.

Indicator Maps

XKB also provides an *indicator map* for each of the thirty-two keyboard indicators; an indicator map specifies:

- The conditions under which the keyboard modifier state affects the indicator.
- The conditions under which the keyboard group state affects the indicator.
- The conditions under which the state of the boolean controls affects the indicator.
- The effect (if any) of attempts to explicitly change the state of the indicator using the core protocol *SetKeyboardControl* request.

If *IM_NoAutomatic* is set in the *flags* field of an indicator map, that indicator never changes in response to changes in keyboard state or controls, regardless of the values for the other fields of the indicator map. If *IM_NoAutomatic* is not set in *flags* , the other fields of the indicator map specify the automatic changes to the indicator in response to changes in the keyboard state or controls.

The *which_groups* and the *groups* fields of an indicator map determine how the keyboard group state affects the corresponding indicator. The *which_groups* field controls the interpretation of *groups* and may contain any one of the following values:

Value	**Interpretation of the Groups Field**
IM_UseNone	The *groups* field and the current keyboard group state are ignored.
IM_UseBase	If *groups* is non-zero, the indicator is lit whenever the base keyboard group is non-zero. If *groups* is zero, the indicator is lit whenever the base keyboard group is zero.
IM_UseLatched	If *groups* is non-zero, the indicator is lit whenever the latched keyboard group is non-zero. If *groups* is zero, the indicator is lit whenever the latched keyboard group is zero.
IM_UseLocked	The *groups* field is interpreted as a mask. The indicator is lit when the current locked keyboard group matches one of the bits that are set in *groups* .
IM_UseEffective	The *groups* field is interpreted as a mask. The indicator is lit when the current effective keyboard group matches one of the bits that are set in *groups* .

The *which_mods* and *mods* fields of an indicator map determine how the state of the keyboard modifiers affect the corresponding indicator. The *mods* field is an XKB modifier definition, as described in Modifier Definitions, which can specify both real and virtual modifiers. The mods field takes effect even if some or all of the virtual indicators specified in *mods* are unbound.

The *which_mods* field can specify one or more components of the XKB keyboard state. The corresponding indicator is lit whenever any of the real modifiers specified in the *mask* field of the *mods* modifier definition are also set in any of the current keyboard state components specified by the *which_mods* . The *which_mods* field may have any combination of the following values:

Value	Keyboard State Component To Be Considered
IM_UseBase	Base modifier state
IM_UseLatched	Latched modifier state
IM_UseLocked	Locked modifier state
IM_UseEffective	Effective modifier state
IM_UseCompat	Modifier compatibility state

The *controls* field specifies a subset of the boolean keyboard controls (see "Boolean" Controls and The EnabledControls Control). The indicator is lit whenever any of the boolean controls specified in *controls* are enabled.

An indicator is lit whenever any of the conditions specified by its indicator map are met, unless overridden by the *IM_NoAutomatic* flag (described above) or an explicit indicator change (described below).

Effects of Explicit Changes on Indicators

If the *IM_NoExplicit* flag is set in an indicator map, attempts to change the state of the indicator are ignored.

If both *IM_NoExplicit* and *IM_NoAutomatic* are both absent from an indicator map, requests to change the state of the indicator are honored but might be immediately superseded by automatic changes to the indicator state which reflect changes to keyboard state or controls.

If the *IM_LEDDrivesKB* flag is set and the *IM_NoExplicit* flag is not, the keyboard state and controls are changed to reflect the other fields of the indicator map, as described in the remainder of this section. Attempts to explicitly change the value of an indicator for which *IM_LEDDrivesKB* is absent or for which *IM_NoExplicit* is present do not affect keyboard state or controls.

The effect on group state of changing an explicit indicator which drives the keyboard is determined by the value of *which_groups* and *groups* , as follows:

which_groups	**New State**	**Effect on Keyboard Group State**
IM_UseNone , or *IM_UseBase*	On or Off	No Effect
IM_UseLatched	On	The *groups* field is treated as a group mask. The keyboard group latch is changed to the lowest numbered group specified in *groups* ; if *groups* is empty, the keyboard group latch is changed to zero.
IM_UseLatched	Off	The *groups* field is treated as a group mask. If the indicator is explicitly extinguished, keyboard group latch is changed to the lowest numbered group not specified in *groups* ; if *groups* is zero, the keyboard group latch is set to the index of the highest legal keyboard group.
IM_UseLocked , or *IM_UseEffective*	On	If the *groups* mask is empty, group is not changed, otherwise the locked keyboard group is changed to the lowest numbered group specified in *groups* .
IM_UseLocked , or *IM_UseEffective*	Off	Locked keyboard group is changed to the lowest numbered group that is not specified in the *groups* mask, or to *Group1* if the *groups* mask contains all keyboard groups.

The effect on the keyboard modifiers of changing an explicit indicator which drives the keyboard is determined by the values that are set in of *which_mods* and *mods* , as follows:

Set in which_mods	**New State**	**Effect on Keyboard Modifiers**
IM_UseBase	On or Off	No Effect
IM_UseLatched	On	Any modifiers specified in the *mask* field of *mods* are added to the latched modifiers.
IM_UseLatched	Off	Any modifiers specified in the *mask* field of *mods* are removed from the latched modifiers.
IM_UseLocked , *IM_UseCompat* , or *IM_UseEffective*	On	Any modifiers specified in the *mask* field of *mods* are added to the locked modifiers.
IM_UseLocked	Off	Any modifiers specified in the *mask* field of *mods* are removed from the locked modifiers.
IM_UseCompat , or *IM_UseEffective*	Off	Any modifiers specified in the *mask* field of *mods* are removed from both the locked and latched modifiers.

Lighting an explicit indicator which drives the keyboard also enables all of the boolean controls specified in the *controls* field of its indicator map. Explicitly extinguishing such an indicator disables all of the boolean controls specified in *controls* .

The effects of changing an indicator which drives the keyboard are cumulative; it is possible for a single change to affect keyboard group, modifiers and controls simultaneously.

If an indicator for which both the *IM_LEDDrivesKB* and *IM_NoAutomatic* flags are specified is changed, the keyboard changes specified above are applied and the indicator is changed to reflect the state that was explicitly requested. The indicator will remain in the new state until it is explicitly changed again.

If the *IM_NoAutomatic* flag is not set for an indicator which drives the keyboard, the changes specified above are applied and the state of the indicator is set to the values specified by the indicator map. Note that it is possible in this case for the indicator to end up in a different state than the one that was explicitly requested. For example, an indicator with *which_mods* of *IM_UseBase* and *mods* of *Shift* is not extinguished if one of the *Shift* keys is physically depressed when the request to extinguish the indicator is processed.

Chapter 10. Keyboard Bells

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 X Keyboard Extension allows clients to disable the audible bell, attach a symbolic name to a bell request or receive an event when the keyboard bell is rung.

Client Notification of Bells

Clients can ask to receive *XkbBellNotify* event when a bell is requested by a client or generated by the server. Bells can be sounded due to core protocol *Bell* requests, X Input Extension *DeviceBell* requests, X Keyboard Extension *XkbBell* requests or for reasons internal to the server such as the XKB *AccessXFeedback* control.

Bell events caused by the *XkbBell* request or by the *AccessXFeedback* control include an optional window and symbolic name for the bell. If present, the window makes it possible to provide some kind of visual indication of which window caused the sound. The symbolic name can report some information about the reason the bell was generated and makes it possible to generate a distinct sound for each type of bell.

Disabling Server Generated Bells

The global *AudibleBell* boolean control for a keyboard indicates whether bells sent to that device should normally cause the server to generate a sound. Applications which provide "sound effects" for the various named bells will typically disable the server generation of bells to avoid burying the user in sounds.

When the *AudibleBell* control is active, all bells caused by core protocol *Bell* and X Input Extension *DeviceBell* requests cause the server to generate a sound, as do all bells generated by the XKB *AccessXFeedback* control. Bells requested via the X *kbBell* request normally cause a server-generated sound, but clients can ask the server not to sound the default keyboard bell.

When the *AudibleBell* control is disabled, the server generates a sound only for bells that are generated using the *XkbBell* request and which specify forced delivery of the bell.

Generating Named Bells

The *XkbBell* request allows clients to specify a symbolic name which is reported in the bell events they cause. Bells generated by the *AccessXFeedback* control of this extension also include a symbolic name, but all kinds of feedback cause a single event even if they sound multiple tones.

The X server is permitted to use symbolic bell names (when present) to generate sounds other than simple tones, but it is not required to do so.

Aside from those used by the XKB *AccessXFeedback* control (see The AccessXFeedback Control), this extension does not specify bell names or their interpretation.

Generating Optional Named Bells

Under some circumstances, some kind of quiet audio feedback is useful, but a normal keyboard bell is not. For example, a quiet "launch effect" can be helpful to let the user know that an application has been started, but a loud bell would simply be annoying.

To simplify generation of these kinds of effects, the *XkbBell* request allows clients to specify "event only" bells. The X server never generates a normal keyboard bell for "event only" bells, regardless of the setting of the global *AudibleBell* control.

If the X server generates different sounds depending bell name, it is permitted to generate a sound even for "event only" bells. This field is intended simply to weed out "normal" keyboard bells.

Forcing a Server Generated Bell

Occasionally, it is useful to force the server to generate a sound. For example, a client could "filter" server bells, generating sound effects for some but sounding the normal server bell for others. Such a client needs a way to tell the server that the requested bell should be generated regardless of the setting of the *AudibleBell* control.

To simplify this process, clients which call the *XkbBell* request can specify that a bell is forced. A forced bell always causes a server generated sound and never causes a *XkbBellNotify* event. Because forced bells do not cause bell notify events, they have no associated symbolic name or event window.

Chapter 11. Keyboard Geometry

The XKB description of a keyboard includes an optional keyboard geometry which describes the physical appearance of the keyboard. Keyboard geometry describes the shape, location and color of all keyboard keys or other visible keyboard components such as indicators. The information contained in a keyboard geometry is sufficient to allow a client program to draw an accurate two-dimensional image of the keyboard.

The components of the keyboard geometry include the following:

- A *symbolic name* to help users identify the keyboard.
- The *width* and *height* of the keyboard, in $\frac{mm}{10}$. For non-rectangular keyboards, the width and height describe the smallest bounding-box that encloses the outline of the keyboard.
- A list of up to *MaxColors* (*32*) *color names* . A color name is a string whose interpretation is not specified by XKB. Other geometry components refer to colors using their indices in this list.
- The *base color* of the keyboard is the predominant color on the keyboard and is used as the default color for any components whose color is not explicitly specified.
- The *label color* is the color used to draw the labels on most of the keyboard keys.
- The *label font* is a string which describes the font used to draw labels on most keys; XKB does not specify a format or name space for font names.
- A list of *geometry properties* . A geometry property associates an arbitrary string with an equally arbitrary name. Geometry properties can be used to provide hints to programs that display images of keyboards, but they are not interpreted by XKB. No other geometry structures refer to geometry properties.
- A list of *key aliases* , as described in Symbolic Names.
- A list of *shapes* ; other keyboard components refer to shapes by their index in this list. A shape consists of a name and one or more closed-polygons called *outlines* . Shapes and outlines are described in detail in Shapes and Outlines.

Unless otherwise specified, geometry measurements are in $\frac{mm}{10}$ units. The origin (0,0) is in the top left corner of the keyboard image. Some geometry components can be drawn rotated; all such objects rotate about their origin in $\frac{1}{10}^{\circ}$ increments.

All geometry components include a *priority* , which indicates the order in which overlapping objects should be drawn. Objects are drawn in order from highest priority (*0*) to lowest (*255*).

The description of the actual appearance of the keyboard is subdivided into named *sections* of related keys and *doodads* . A a *doodad* describes some visible aspect of the keyboard that is not a key. A section is a collection of keys and doodads that are physically close together and logically related.

Shapes and Outlines

An outline is a list of one or more points which describes a single closed-polygon, as follows:

- A list with a single point describes a rectangle with one corner at the origin of the shape (*0* , *0*) and the opposite corner at the specified point.
- A list of two points describes a rectangle with one corner at the position specified by the first point and the opposite corner at the position specified by the second point.
- A list of three or more points describes an arbitrary polygon. If necessary, the polygon is automatically closed by connecting the last point in the list with the first.
- A non-zero value for the *cornerRadius* field specifies that the corners of the polygon should be drawn as circles with the specified radius.

All points in an outline are specified relative to the origin of the enclosing shape. Points in an outline may have negative values for the X and Y coordinate.

One outline (usually the first) is the primary outline; a keyboard display application can generate a simpler but still accurate keyboard image by displaying only the primary outlines for each shape. Non-rectangular keys must include a rectangular *approximation* as one of the outlines associated with the shape; the approximation is not normally displayed but can be used by very simple keyboard display applications to generate a recognizable but degraded image of the keyboard.

Sections

Each section has its own coordinate system — if a section is rotated, the coordinates of any components within the section are interpreted relative to the edges that were on the top and left before rotation. The components that make up a section include:

- A list of *rows* . A row is a list of horizontally or vertically adjacent keys. Horizontal rows parallel the (pre-rotation) top of the section and vertical rows parallel the (pre-rotation) left of the section. All keys in a horizontal row share a common top coordinate; all keys in a vertical row share a left coordinate.

  A key description consists of a key *name* , a *shape* , a key *color* , and a *gap* . The key *name* should correspond to one of the keys named in the keyboard names description, the *shape* specifies the appearance of the key, and the key *color* specifies the color of the key (not the label on the key). Keys are normally drawn immediately adjacent to one another from left-to-right (or top-to-bottom) within a row. The *gap* field specifies the distance between a key and its predecessor.

- An optional list of doodads; any type of doodad can be enclosed within a section. Position and angle of rotation are relative to the origin and angle of rotation of the sections that contain them. Priority is relative to the other components of the section, not to the keyboard as a whole.
- An optional list of *overlay keys* . Each overlay key definition indicates a key that can yield multiple scan codes and consists of a field named *under* , which specifies the primary name of the key and a field named *over* , which specifies the name for the key when the overlay keycode is selected. The key specified in *under* must be a member of the section that contains the overlay key definition, while the key specified in over must not.

Doodads

Doodads can be global to the keyboard or part of a section. Doodads have symbolic names of arbitrary length. The only doodad name whose interpretation is specified by XKB is "Edges", which describes the outline of the entire keyboard, if present.

All doodads report their origin in fields named *left* and *top* . XKB supports five kinds of doodads:

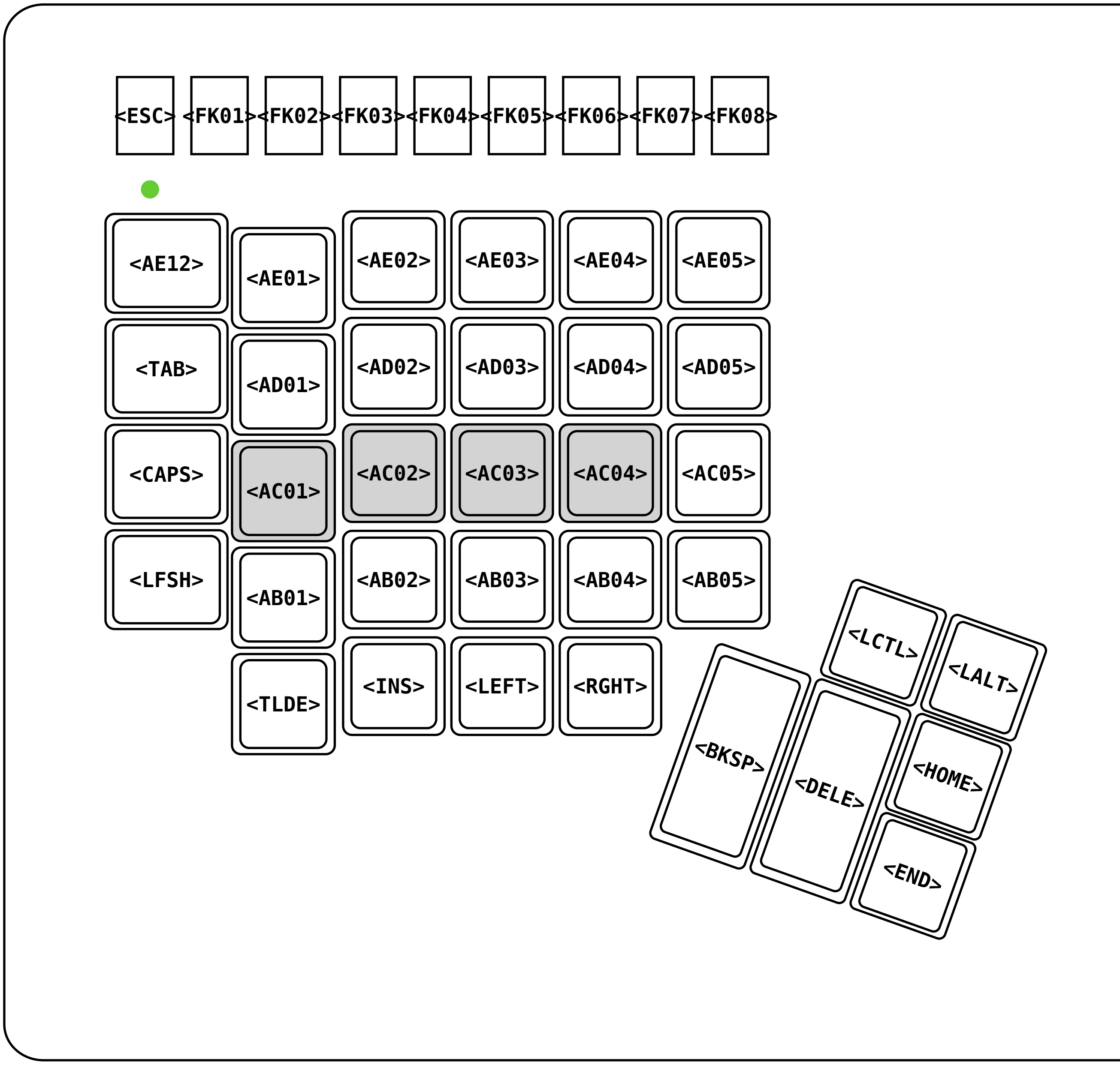
- An *indicator doodad* describes one of the physical keyboard indicators. Indicator doodads specify the shape of the indicator, the indicator color when it is lit (*on_color*) and the indicator color when it is dark (*off_color*).
- An *outline doodad* describes some aspect of the keyboard to be drawn as one or more hollow, closed polygons. Outline doodads specify the shape, color, and angle of rotation about the doodad origin at which they should be drawn.
- A *solid doodad* describes some aspect of the keyboard to be drawn as one or more filled polygons. Solid doodads specify the shape, color and angle of rotation about the doodad origin at which they should be drawn.
- A *text doodad* describes a text label somewhere on the keyboard. Text doodads specify the label string, the font and color to use when drawing the label, and the angle of rotation of the doodad about its origin.
- A *logo doodad* is a catch-all, which describes some other visible element of the keyboard. A logo doodad is essentially an outline doodad with an additional symbolic name that describes the element to be drawn.

  If a keyboard display program recognizes the symbolic name, it can draw something appropriate within the bounding region of the shape specified in the doodad. If the symbolic name does not describe a recognizable image, it should draw an outline using the specified shape, outline, and angle of rotation.

  The XKB extension does not specify the interpretation of logo names.

Keyboard Geometry Example

Consider the following example keyboard:



This keyboard has six sections: The left and right function sections (at the very top) each have one horizontal row with eight keys. The left and right alphanumeric sections (the large sections in the middle) each have six vertical rows, with four or five keys in each row. The left and right editing sections each have three vertical rows with one to three keys per row; the left editing section is rotated 20° clockwise about its origin while the right editing section is rotated 20° counterclockwise.

This keyboard has four global doodads: Three small, round indicators and a rectangular logo. The program which generated this image did not recognize the logo, so it displays an outline with an appropriate shape in its place.

This keyboard has seven shapes: All of the keys in the two function sections use the "FKEY" shape. Most of the keys in the alphanumeric sections, as well as four of the keys in each of the editing sections use the "NORM" shape. The keys in the first column of the left alphanumeric section and the last column of the right alphanumeric section all use the "WIDE" shape. Two keys in each of the editing sections use the "TALL" shape. The "LED" shape describes the three small, round indicators between the function and alphabetic sections. The "LOGO" shape describes the keyboard logo, and the "EDGE" shape describes the outline of the keyboard as a whole.

The keyboard itself is white, as are all of the keys except for the eight keys that make up the home row, which use the "grey20" color. It isn’t really visible in this picture, but the three indicators have an "on" color of "green" and are "green30" when they are turned off. The keys in the alphanumeric and editing sections all have a (vertical) gap of 0.5mm; the keys in the two function sections have a (horizontal) gap of 3mm.

Many of the keys in the right alphanumeric section, and the rightmost key in the right editing section are drawn with two names in this image. Those are overlay keys; the bottom key name is the normal name while the overlay name is printed at the top. For example, the right editing section has a single overlay key entry, which specifies an *under* name of *<SPCE>* and an *over* name of *<KP0>* , which indicates that the key in question is usually the shift key, but can behave like the *0* key on the numeric keypad when an overlay is active.

Chapter 12. Interactions Between XKB and the Core Protocol

In addition to providing a number of new requests, XKB 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 today.

There are a few 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, XKB includes ways to specify the correspondence between core protocol and XKB modifiers and symbols.

This section describes the differences between the core X protocol's notion of a keyboard mapping and XKB and explains the ways they can interact.

Group Compatibility Map

As described in Keyboard State, the current keyboard group is reported to XKB-aware clients in bits 13-14 of the state field of many core protocol events. XKB-unaware clients cannot interpret those bits, but they might use a keyboard modifier to implement support for a single keyboard group. To ensure that pre-XKB clients continue to work when XKB is present, XKB makes it possible to map an XKB state field, which includes both keyboard group and modifier state into a pre-XKB state field which contains only modifiers.

A keyboard description includes one *group compatibility map* per keyboard group (four in all). Each such map is a modifier definition (i.e. specifies both real and virtual modifiers) which specifies the modifiers to be set in the compatibility states when the corresponding keyboard group is active. Here are a few examples to illustrate the application of the group compatibility map:

Group	GroupCompat Map	Effective Modifiers	State for XKB Clients	Compatibility Modifiers	State for non-XKB Clients
1	Group1=None	Shift	x00xxxxx00000001	Shift	xxxxxxxx00000001
2	Group2=Mod3	None	x01xxxxx00000000	Mod3	xxxxxxxx00100000
3	Group3=Mod2	Shift	x10xxxxx00000001	Shift +Mod2	xxxxxxxx00010001
4	Group4=None	Control	x11xxxxx00000100	Control	xxxxxxxx00000100

Note that non-XKB clients (i.e. clients that are linked with a version of the X library that does not support XKB) cannot detect the fact that *Group4* is active in this example because the group compatibility map for *Group4* does not specify any modifiers.

Setting a Passive Grab for an XKB State

The fact that the *state* field of an event might look different when XKB is present can cause problems with passive grabs. Existing clients specify the modifiers they wish to grab using the rules defined by the core protocol, which use a normal modifier to indicate keyboard group. If we used an XKB state field, the high bits of the state field would be non-zero whenever the keyboard was in any group other than *Group1* , and none of the passive grabs set by clients could ever be triggered.

To avoid this behavior, the X server normally uses the compatibility grab state to decide whether or not to activate a passive grab, even for XKB-aware clients. The group compatibility map attempts to encode the keyboard group in one or more modifiers of the compatibility state, so existing clients continue to work exactly the way they do today. By default, there is no way to directly specify a keyboard group in a *Grabbed* or *GrabButton* request, but groups can be specified indirectly by correctly adjusting the group compatibility map.

Clients that wish to specify an XKB keyboard state, including a separate keyboard group, can set the *GrabsUseXKBState* per-client flag which indicates that all subsequent key and button grabs from the requesting clients are specified using an XKB state.

Whether the XKB or core state should be used to trigger a grab is determined by the setting of the *GrabsUseXKBState* flag for the requesting client at the time the key or button is grabbed. There is no way to change the state to be used for a grab that is already registered or for grabs that are set by some other client.

Changing the Keyboard Mapping Using the Core Protocol

An XKB keyboard description includes a lot of information that is not present in the core protocol description of a keyboard. Whenever a client remaps the keyboard using core protocol requests, XKB examines the map to determine likely default values for the components that cannot be specified using the core protocol.

Some aspects of this automatic mapping are configurable, and make it fairly easy to take advantage of many XKB features using existing tools like *xmodmap* , but much 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.

Explicit Keyboard Mapping Components

This automatic remapping might accidentally replace definitions that were explicitly requested by an application, so the XKB keyboard description defines a set of *explicit components* for each key; any components that are listed in the explicit components for a key are not changed by the automatic keyboard mapping. The explicit components field for a key can contain any combination of the following values:

Bit in Explicit Mask	**Protects Against**
ExplicitKeyType1	Automatic determination of the key type associated with *Group1* (see Assigning Types To Groups of Symbols for a Key)
ExplicitKeyType2	Automatic determination of the key type associated with *Group2* (see Assigning Types To Groups of Symbols for a Key)
ExplicitKeyType3	Automatic determination of the key type associated with *Group3* (see Assigning Types To Groups of Symbols for a Key).
ExplicitKeyType4	Automatic determination of the key type associated with *Group4* (see Assigning Types To Groups of Symbols for a Key).
ExplicitInterpret	Application of any of the fields of a symbol interpretation to the key in question (see Assigning Actions To Keys).
ExplicitAutoRepeat	Automatic determination of autorepeat status for the key, as specified in a symbol interpretation (see Assigning Actions To Keys).
ExplicitBehavior	Automatic assignment of the *KB_Lock* behavior to the key, if the *LockingKey* flag is set in a symbol interpretation (see Assigning Actions To Keys).
ExplicitVModMap	Automatic determination of the virtual modifier map for the key based on the actions assigned to the key and the symbol interpretations which match the key (see Assigning Actions To Keys).

Assigning Symbols To Groups

The first step in applying the changes specified by a core protocol *ChangeKeyboardMapping* request to the XKB description of a keyboard is to determine the number of groups that are defined for the key and the width of each group. The XKB extension does not change key types in response to core protocol *SetModifierMapping* requests, but it does choose key actions as described in Assigning Actions To Keys.

Determining the number of symbols required for each group is straightforward. If the key type for some group is not protected by the corresponding *ExplicitKeyType* component, that group has two symbols. If any of the explicit components for the key include *ExplicitKeyType3* or *ExplicitKeyType4* , the width of the key type currently assigned to that group determines the number of symbols required for the group in the core protocol keyboard description. The explicit type components for *Group1* and *Group2* behave similarly, but for compatibility reasons the first two groups must have at least two symbols in the core protocol symbol mapping. Even if an explicit type assigned to either of the first two keyboard groups has fewer than two symbols, XKB requires two symbols for it in the core keyboard description.

If the core protocol request contains fewer symbols than XKB needs, XKB adds trailing *NoSymbol* keysyms to the request to pad it to the required length. If the core protocol request includes more symbols than it needs, XKB truncates the list of keysyms to the appropriate length.

Finally, XKB divides the symbols from the (possibly padded or truncated) list of symbols specified by the core protocol request among the four keyboard groups. In most cases, the symbols for each group are taken from the core protocol definition in sequence (i.e. the first pair of symbols is assigned to *Group1* , the second pair of symbols is assigned to *Group2* , and so forth). If either *Group1* or *Group2* has an explicitly defined key type with a width other than two, it gets a little more complicated.

Assigning Symbols to Groups One and Two with Explicitly Defined Key Types

The server assigns the first four symbols from the expanded or truncated map to the symbol positions *G1L1* , *G1L2*, *G2L1* and *G2L2*, respectively. If the key type assigned to *Group1* reports more than two shift levels, the fifth and following symbols contain the extra keysyms for *Group2* . If the key type assigned to *Group2* reports more than two shift levels, the extra symbols follow the symbols (if any) for *Group1* in the core protocol list of symbols. Symbols for *Group3* and *Group4* are contiguous and follow the extra symbols, if any, for *Group1* and *Group2* .

For example, consider a key with a key type that returns three shift levels bound to each group. The symbols bound to the core protocol are assigned in sequence to the symbol positions:

```
G1L1, G1L2, G2L1, G2L2, G1L3, G2L3, G3L1, G3L2, G3L3, G4L1, G4L2, and G4L3
```

For a key with a width one key type on group one, a width two key type on group two and a width three key type on group three, the symbols bound to the key by the core protocol are assigned to the following key positions:

```
G1L1, (G1L2), G2L1, G2L2, G3L1, G3L2, G3L3
```

Note that the second and fourth symbols (positions *G1L2 and G2L2*) can never be generated if the key type associated with the group yields only one symbol. XKB accepts and ignores them in order to maintain compatibility with the core protocol.

Assigning Types To Groups of Symbols for a Key

Once the symbols specified by *ChangeKeyboardMapping* have been assigned to the four keyboard groups for a key, the X server assigns a key type to each group on the key from a canonical list of key types. The first four 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 and non-alphabetic keys that have exactly two symbols per group. By default, the *TWO_LEVEL* type yields column *1* if the Shift modifier is set, column *0* otherwise. Index *1* in any key symbol map specifies key type *TWO_LEVEL* .
ALPHABETIC	Describes alphabetic keys that have exactly two symbols per group. The default definition of the *ALPHABETIC* type provides shift-cancels-caps behavior as described in Key Types. Index *2* in any key symbol map specifies key type *ALPHABETIC* .
KEYPAD	Describes numeric keypad keys with two symbols per group. Yields column *1* if either of the *Shift* modifier or the real modifier bound to the virtual modifier named *NumLock* are set. Yields column *0* if neither or both modifiers are set. Index *3* in any key symbol map specifies key type *KEYPAD* .

Users or applications may change 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.

Before assigning key types to groups, the X server expands any alphanumeric symbol definitions as follows:

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. For the purposes of this expansion, XKB ignores the locale and uses the capitalization rules defined in Default Symbol Transformations.

For each keyboard group that does not have an explicit type definition, XKB chooses a key type from the canonical key types. If the second symbol assigned to a group is *NoSymbol* (after alphabetic expansion), the server assigns key type *ONE_LEVEL* . If the group contains the lowercase and uppercase forms of a single glyph (after alphanumeric expansion), the server assigns key type *ALPHABETIC* . If either of the symbols in a group is a numeric keypad keysym (*KP_**), the server assigns key type *KEYPAD* . Otherwise, it assigns key type *TWO_LEVEL* .

Finally, XKB determines the number of groups of symbols that are actually defined for the key. Trailing empty groups (i.e. groups that have *NoSymbol* in all symbol positions) are ignored.

There are two last special cases for compatibility with the core protocol: If, after trailing empty groups are excluded, all of the groups of symbols bound to the key have identical type and symbol bindings, XKB assigns only one group to the key. If *Group2* is empty and either of *Group3* or *Group4* are not, and if neither *Group1* nor *Group2* have explicit key types, XKB copies the symbols and key type from *Group1* into *Group2* .

Assigning Actions To Keys

Once symbols have been divided into groups and key types chosen for the keys affected by a *ChangeKeyboardMapping* request, XKB examines the symbols and modifier mapping for each changed key and assigns server actions where appropriate. XKB also automatically assigns server actions to changed keys if the client issues a core protocol *SetModifierMapping* request, and does so optionally in response to *XkbSetMap* and *XkbSetCompatMap* requests.

The compatibility map includes a list of *symbol interpretations* , which XKB compares to each symbol associated with any changed keys in turn, unless the *ExplicitInterp* component is set for a key. Setting the *ExplicitInterp* component prevents the application of symbol interpretations to that key.

If the modifiers and keysym specified in a symbol interpretation match the modifier mapping and a symbol bound to a changed key that is not protected by *ExplicitInterp* , the server applies the symbol interpretation to the symbol position. 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.

XKB uses four of the fields of a symbol interpretation to decide if it matches one of the symbols bound to some changed key:

- The *symbol* field is a keysym which matches if it has the value *NoSymbol* or is identical to the symbol in question.
- The modifiers specified in the *mods* field are compared to the modifiers affected by the key in question as indicated by *match* .
- The *match* field can specify any of the comparisons: *NoneOf* , *AnyOfOrNone* , *AnyOf* , *AllOf* or *Exactly* .
- The *levelOneOnly* setting, indicates that the interpretation in question should only use the modifiers bound to this key by the modifier mapping if the symbol that matches in level one of its group. Otherwise, if the symbol being considered is not in shift level one of its group, the server behaves as if the modifier map for the key were empty. Note that it is still possible for such an interpretation to apply to a symbol in a shift level other than one if it matches a key without modifiers; the *levelOneOnly* flag only controls the way that matches are determined and that the key modifiers are applied when an interpretation does match.

Applying a symbol interpretation can affect several aspects of the XKB definition of the key symbol mapping to which it is applied:

- The *action* specified in the symbol interpretation is bound to the symbol position; any key event which yields that symbol will also activate the new action.
- If the matching symbol is in position G1L1, the autorepeat behavior of the key is set from the *autorepeat* field of the symbol interpretation. The *ExplicitAutoRe-*

peat component protects the autorepeat status of a key from symbol interpretation initiated changes.

- If the symbol interpretation specifies an associated virtual modifier, that virtual modifier is added to the virtual modifier map for the key. The *ExplicitVModMap* component guards the virtual modifier map for a key from automatic changes. If the *levelOneOnly* flag is set for the interpretation, and the symbol in question is not in position G1L1, the virtual modifier map is not updated.
- If the matching symbol is in position G1L1, and the *locking key* field is set in the symbol interpretation, the behavior of the key is changed to *KB_Lock* (see Key Behavior). The *ExplicitBehavior* component prevents this change.

If no interpretations match a given symbol or key, the server uses: *SA_NoAction* , autorepeat enabled, non-locking key. with no virtual modifiers.

If all of the actions computed for a key are *SA_NoAction* , the server assigns an length zero list of actions to the key.

If the core protocol modifier mapping is changed, the server regenerates actions for the affected keys. The *XkbSetMap* and *XkbSetCompatMap* requests can also cause actions for some or all keyboard keys to be recomputed.

Updating Everything Else

Changes to the symbols or modifier mapping can affect the bindings of virtual modifiers. If any virtual modifiers change, XKB updates all of its data structures to reflect the change. Applying virtual modifier changes to the keyboard mapping night result in changes to types, the group compatibility map, indicator maps, internal modifiers or ignore locks modifiers.

Effects of XKB on Core Protocol 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 components of the keyboard state are reported to XKB-aware clients depending on context (see Keyboard State for a detailed description of each of the keyboard state components):

- The effective modifier state is reported in *XkbStateNotify* events and in response to *XkbGetState* requests.
- The symbol lookup state is reported to XKB-aware clients in the state field of core protocol and input extension key press and release events that do not activate passive grabs. Unless the *LookupStateWhenGrabbed* per-client flag is set, the lookup state is only reported in these events when no grabs are active.
- The grab state 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 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 the *groupsWrap* flag for the keyboard.

The server reports compatibility states to any clients that have not issued a successful *XkbUseExtension* 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 whenever an XKB-aware client would receive the XKB lookup state. The compatibility grab state is reported to XKB-unaware clients whenever an XKB client would receive the XKB grab state.

If the *GrabsUseXKBState* per-client option is not set, even XKB-aware clients receive the compatibility grab state in events that trigger or terminate passive grabs. If this flag is not set, XKB clients also receive the compatibility grab or lookup state whenever any keyboard grab is active.

If the *LookupStateWhenGrabbed* per-client option is set, clients receive either the XKB or compatibility lookup state when the keyboard is grabbed, otherwise they receive either the XKB or compatibility grab state. All non-XKB clients receive the compatibility form of the appropriate state component; the form that is sent to an XKB-aware client depends on the setting of the *GrabsUseXKBState* option for that client.

Effect of XKB on Core Protocol Requests

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.

The symbols associated with the XKB keyboard description appear in the order:

```
G1L1 G1L2 G2L1 G2L2 G1L3-n G2L3-n G3L* G4L*
```

If the type associated with *Group1* is width one, the second symbol is *NoSymbol* ; if the type associated with *Group2* is width one, the fourth symbol is *NoSymbol* .

If a key has only one group but the keyboard has several, the symbols for *Group1* are repeated for each group. For example, given a keyboard with three groups and a key with one group that contains the symbols { *a A* }, the core protocol description would contain the six symbols: { *a A a A a A* }. As a slightly more complicated example, an XKB key which had a single width three group with the symbols { *a b c* } would show up in the generated core protocol keyboard description with the symbols { *a b a b c c a b c* } for a keyboard with three groups.

The generated modifier mapping for a key contains all of the modifiers affected by all of the actions associated with the key plus all of the modifiers associated with any virtual modifiers bound to the key by the virtual modifier mapping. If any of the actions associated with a key affect any component of the keyboard group, any modifiers specified in any entry of the group compatibility map (see Group Compatibility Map) 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 *XkbMapNotify* to clients that have explicitly selected them and core protocol *MappingNotify* events to clients that have not. Once a client requests *XkbMapNotify* events, the server stops sending it *MappingNotify* events to inform it of keyboard changes.

Sending Events to Clients

XKB normally assumes that events sent to clients using the core protocol *SendEvent* request contain a core protocol state, if applicable. If the client which will receive the event is not XKB-capable, XKB attempts to convert the core state to an XKB state as follows: if any of the modifiers bound to *Group2* in the group compatibility map are set in the event state, XKB clears them in the resulting event but sets the effective group in the event state to *Group2* .

If the *PCF_SendEventUsesXKBState* per-client flag is set at the time of the SendEvent request, XKB instead assumes that the event reported in the event is an XKB state. If the receiving client is not XKB-aware, the extension converts the XKB state (which contains the effective state in bits 13-14) to a core state by applying the group compatibility map just as it would for actual key events.

Chapter 13. The Server Database of Keyboard Components

The X server maintains a database of keyboard components and common keyboard mappings. This database contains five kinds of components; when combined, these five components provide a complete description of a keyboard and its behavior.

The X Keyboard Extension provides requests to list the contents of this database, to assemble and complete keyboard descriptions by merging the current keyboard description with the contents of this database, or to replace the current keyboard description with a complete keyboard description assembled as described below.

Component Names

Component and keymap names have the form " *class* (*member*)" where *class* describes a subset of the available components for a particular type and the optional *member* identifies a specific component from that subset. For example, the name "atlantis(acme)" might specify the symbols used for the atlantis national keyboard layout by the vendor "acme." Each class has an optional *default* member — references which specify a class but not a member refer to the default member of the class, if one exists.

The *class* and *member* names are both specified using characters from the Latin-1 character set. XKB implementations must accept all alphanumeric characters, minus ('-') and underscore ('_') in class or member names, and must not accept parentheses, plus, vertical bar, percent sign, asterisk, question mark or white space. The use of other characters is implementation-dependent.

Partial Components and Combining Multiple Components

Some of the elements in the server database contain describe only a piece of the corresponding keyboard component. These *partial* components should be combined with other components of the same type to be useful.

For example, a partial symbols map might describe the differences between a common ASCII keyboard and some national layout. Such a partial map is not useful on its own because it does not include those symbols that are the same on both the ASCII and national layouts (such as function keys). On the other hand, this partial map can configure *any* ASCII keyboard to use a national layout.

Two components can be combined in two ways:

- If the second component *overrides* the first, any definitions that are present in both components are taken from the second.
- If the second component *augments* the first, any definitions that are present in both components are taken from the first.

Applications can use a *component expression* to combine multiple components of some time into a complete description of some aspect of the keyboard. A component expression is a string which lists the components to be combined separated by operators which specify the rules for combining them. A complete description is assembled from the listed components, left to right, as follows:

- If the new elements are being merged with an existing map, the special component name '%' refers to the unmodified value of the map.
- The '+' operator specifies that the next specified component should override the current assembled definition.
- The '|' operator specifies that the next specified component should augment the currently assembled definition.
- If the new elements are being merged with an existing map and the component expression begins with an operator, a leading '%' is implied.
- If any unknown or illegal characters appear anywhere in the string, the entire expression is invalid and is ignored.

For example, the component expression "+de" specifies that the default element of the "de" map should be applied to the current keyboard mapping, overriding any existing definitions.

A slightly more involved example: the expression "acme(ascii)+de(basic)|iso9995-3" constructs a German (de) mapping for the ASCII keyboard supplied by the "acme" vendor. The new definition begins with the symbols for the default ASCII keyboard for Acme, overrides them with any keys that are defined for the default German keyboard layout and then applies the definitions from the iso9995-3 to any undefined keys or groups of keys (part three of the iso9995 standard defines a common set of bindings for the secondary group, but allows national layouts to override those definitions where necessary).

Component Hints

Each component has a set of flags that provide some additional hints about that component. XKB provides these hints for clients that present the keyboard database to users and specifies their interpretation only loosely. Clients can use these hints to constrain the list of components or to control the way that components are presented to the user.

Hints for a component are reported with its name. The least significant byte of the hints field has the same meaning for all five types of keyboard components, and can contain any combination of the following values:

Flag	**Meaning**
LC_Hidden	Indicates a component that should not normally be presented to the user.
LC_Default	Indicates a component that is the default member of its class.
LC_Partial	Indicates a partial component.

The interpretation of the most significant byte of the hints field is dependent on the type of component. The hints defined for each kind of component are listed in the section below that describes that kind of component.

Keyboard Components

The five types of components stored in the server database of keyboard components correspond to the *symbols* , *geometry* , *keycodes* , *compat* and *types* symbolic names associated with a keyboard.

The Keycodes Component

The *keycodes* component of a keyboard mapping specifies the range and interpretation of the raw keycodes reported by the device. It sets the *keycodes* symbolic name, the minimum and maximum legal keycodes for the keyboard, and the symbolic name for each key. The keycodes component might also contain aliases for some keys, symbolic names for some indicators, and a description of which indicators are physically present.

The special keycodes component named "computed" indicates that XKB should assign unused keycodes to any unknown keys referenced by name by any of the other components. The computed keycodes component is useful primarily when browsing keymaps because it makes it possible to use the symbols and geometry components without having to find a set of keycodes that includes keycode definitions for all of the keys listed in the two components.

XKB defines no hints that are specific to the keycodes component.

The Types Component

The *types* component of a keyboard mapping specifies the key types that can be associated with the various keyboard keys. It affects the *types* symbolic name and the list of types associated with the keyboard (see Key Types). The types component of a keyboard mapping can also optionally contain real modifier bindings and symbolic names for one or more virtual modifiers.

The special types component named "canonical" always contains the types and definitions listed in Canonical Key Types of this document.

XKB defines no hints that are specific to the types component.

The Compatibility Map Component

The *compatibility map* component of a keyboard mapping primarily specifies the rules used to assign actions to keysyms. It affects the *compat* symbolic name, the symbol compatibility map and the group compatibility map. The compat component might also specify maps for some indicators and the real modifier bindings and symbolic names of some virtual modifiers.

XKB defines no hints that are specific to the compatibility map component.

The Symbols Component

The *symbols* component of a keyboard mapping specifies primarily the symbols bound to each keyboard key. It affects the *symbols* symbolic name, a key symbol mapping for each key, they keyboard modifier mapping, and the symbolic names for the keyboard symbol groups. Optionally, the *symbols* component can contain explicit actions and behaviors for some keys, or the real modifier bindings and symbolic names for some virtual modifiers.

XKB defines the following additional hints for the symbols component:

Flag	**Meaning**
LC_AlphanumericKeys	Indicates a symbol component that contains bindings primarily for an alphanumeric section of the keyboard.
LC_ModifierKeys	Indicates a symbol component that contains bindings primarily for modifier keys.
LC_KeypadKeys	Indicates a symbol component that contains bindings primarily for numeric keypad keys.
LC_FunctionKeys	Indicates a symbol component that contains bindings primarily for function keys.
LC_AlternateGroup	Indicates a symbol component that contains bindings for an alternate keyboard group.

These hints only apply to partial symbols components; full symbols components are assumed to specify all of the pieces listed above.

Note

The alphanumeric, modifier, keypad or function keys hints should describe the primary intent of the component designer and should not simply an exhaustive list of the kinds of keys that are affected. For example, national keyboard layouts affect primarily alphanumeric keys, but many affect a few modifier keys too; such mappings should set only *LC_AlphanumericKeys* hint. In general, symbol components should set only one of those four flags (though *LC_AlternateGroup* may be combined with any of the other flags).

The Geometry Component

The *geometry* component of a keyboard mapping specifies primarily the geometry of the keyboard. It contains the geometry symbolic name and the keyboard geometry description. The geometry component might also contain aliases for some keys or symbolic names for some indicators and might affect the set of indicators that are physically present. Key aliases defined in the geometry component of a keyboard mapping override those defined in the keycodes component.

XKB defines no hints that are specific to the geometry component.

Complete Keymaps

The X server also reports a set of fully specified keymaps. The keymaps specified in this list are usually assembled from the components stored in the rest of the database and typically represent the most commonly used keymaps for a particular system.

XKB defines no hints that are specific to complete keymaps.

Chapter 14. Replacing the Keyboard "On-the-Fly"

XKB supports the *XkbNewKeyboardNotify* event, which reports a change in keyboard geometry or the range of supported keycodes. The server can generate an *XkbNewKeyboardNotify* event when it detects a new keyboard, or in response to an *XkbGetKeyboardByName* request (see Using the Server's Database of Keyboard Components) which loads a new keyboard description.

When a client opens a connection to the X server, the server reports the minimum and maximum keycodes. If the range of supported keycodes is changed, XKB keeps track of the minimum and maximum keycodes that were reported to each client and filters out any events that fall outside of that range. Note that these events are simply ignored; they are not delivered to some other client.

When the server sends an *XkbNewKeyboardNotify* event to a client to inform it of the new keycode range, XKB resets the stored range of legal keycodes to the keycode range reported in the event. Non-XKB clients and XKB-aware clients that do not request *XkbNewKeyboardNotify* events never receive events from keys that fall outside of the legal range that XKB maintains for that client.

When a client requests *XkbNewKeyboardNotify* events, the server compares the range of keycodes for the current keyboard to the range of keycodes that are valid for the client. If they are not the same, the server immediately sends that client an *XkbNewKeyboardNotify* event. Even if the "new" keyboard is not new to the server, it is new to this particular client.

In addition to filtering out-of-range key events, XKB:

- Adjusts core protocol *MappingNotify* events to refer only to keys that match the stored legal range.
- Reports keyboard mappings for keys that match the stored legal range to clients that issue a core protocol *GetKeyboardMapping* request.
- Reports modifier mappings only for keys that match the stored legal range to clients that issue a core protocol *GetModifierMapping* request.
- Restricts the core protocol *ChangeKeyboardMapping* and *SetModifierMapping* requests to keys that fall inside the stored legal range.

In short, XKB does everything possible to hide the fact that the range of legal keycodes has changed from clients non-XKB clients, which cannot be expected to deal with it. The corresponding XKB events and requests do *not* pay attention to the legal keycode range in the same way because XKB makes it possible for clients to track changes to the keycode range for a device and respond to them.

Chapter 15. Interactions Between XKB and the X Input Extension

All XKB interactions with the input extension are optional; implementors are free to restrict the effects of the X Keyboard Extension to the core keyboard device. The *XkbGetExtensionDeviceInfo* request reports whether or not an XKB implementation supports a particular capability for input extension devices.

XKB recognizes the following interactions with the X Input Extension:

Name	**Capability**
XI_Keyboards	If set, applications can use all XKB requests and events with extension keyboards.
XI_ButtonActions	If set, clients can assign key actions to buttons, even on input extension devices that are not keyboards.
XI_IndicatorNames	If set, clients can assign names to indicators on non-keyboard extension devices.
XI_IndicatorMaps	If set, clients can assign indicator maps to indicators on non-keyboard extension devices.
XI_IndicatorState	If set, clients can change the state of device indicators using the *XkbSetExtensionDeviceInfo* request.

Attempts to use an XKB feature with an extension device fail with a *Keyboard* error if the server does not support the *XkbXI_Keyboards* optional feature. If a capability particular capability other than *XkbXI_Keyboards* is not supported, attempts to use it fail silently. The replies for most requests that can use one of the other optional features include a field to report whether or not the request was successful, but such requests do not cause an error condition.

Clients can also request an *XkbExtensionDeviceNotify* event. This event notifies interested clients of changes to any of the supported XKB features for extension devices, or if a request from the client that is receiving the event attempted to use an unsupported feature.

Using XKB Functions with Input Extension Keyboards

All XKB requests and events include a device identifier which can refer to an input extension *KeyClass* device, if the implementation allows XKB to control extension devices. If the implementation does not support XKB manipulation of extension devices, the device identifier is ignored but it must be either *0* or *UseCoreKbd* .

Implementations which do not support the use of XKB functions with extension keyboards must not set the *XkbXI_Keyboards* flag. Attempts to use XKB features on an extension keyboard with an implementation that does not support this feature yield a *Keyboard* error.

Pointer and Device Button Actions

The XKB extension optionally allows clients to assign any key action (see Key Actions) to core pointer or input extension device buttons. This makes it possible to control the keyboard or generate keyboard key events from extension devices or from the core pointer.

XKB implementations are required to support actions for the buttons of the core pointer device, but support for actions on extension devices is optional. Implementations which do not support button actions for extension devices must not set the *XkbXI_ButtonActions* flag.

Attempts to query or assign button actions with an implementation that does not support this feature report failure in the request reply and might cause the server to send an *XkbExtensionDeviceNotify* event to the client which issued the request that failed. Such requests never cause an error condition.

Indicator Maps for Extension Devices

The XKB extension allows applications to assign indicator maps to the indicators of non-keyboard extension devices. If supported, maps can be assigned to all extension device indicators, whether they are part of a keyboard feedback or part of an indicator feedback.

Implementations which do not support indicator maps for extension devices must not set the *XkbXI_IndicatorMaps* flag.

Attempts to query or assign indicator maps with an implementation that does not support this feature report failure in the request reply and might cause the server to send an *XkbExtensionDeviceNotify* event to the client which issued the request that failed. Such requests never cause an error condition.

If this feature is supported, the maps for the default indicators on the core keyboard device are visible both as extension indicators and as the core indicators. Changes made with *XkbSetDeviceInfo* are visible via *XkbGetIndicatorMap* and changes made with *XkbSetIndicatorMap* are visible via *XkbGetDeviceInfo* .

Indicator Names for Extension Devices

The XKB extension allows applications to assign symbolic names to the indicators of non-keyboard extension devices. If supported, symbolic names can be assigned to all extension device indicators, whether they are part of a keyboard feedback or part of an indicator feedback.

Implementations which do not support indicator maps for extension devices must not set the *XkbXI_IndicatorMaps* flag.

Attempts to query or assign indicator names with an implementation that does not support this feature report failure in the request reply and might cause the server to send an *XkbExtensionDeviceNotify* event to the client which issued the request that failed. Such requests never cause an error condition.

If this feature is supported, the names for the default indicators on the core keyboard device are visible both as extension indicators and as the core indicators.

Changes made with *XkbSetDeviceInfo* are visible via *XkbGetNames* and changes made with *XkbSetNames* are visible via *XkbGetDeviceInfo* .

Chapter 16. XKB Protocol Requests

This document uses the syntactic conventions and common types defined by the specification of the core X protocol with a number of additions, which are detailed below.

Errors

If a client attempts to use any other XKB request except *XkbUseExtension* before the extension is properly initialized, XKB reports an *Access* error and ignores the request. XKB is properly initialized once *XkbUseExtension* reports that the client has asked for a supported or compatible version of the extension.

Keyboard Errors

In addition to all of the errors defined by the core protocol, 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 an appropriate. Unless otherwise noted, any request with an argument of type KB_DEVICESPEC can cause *Keyboard* errors if an illegal or inappropriate device is specified.

When the extension reports a Keyboard error, the most significant byte of the *resource_id* is a further refinement of the error cause, as defined in the table below. The least significant byte contains the device, class, or feedback id as indicated:

high-order byte	value	meaning	low-order byte
XkbErr_BadDevice	0xff	device not found	device id
XkbErr_BadClass	0xfe	device found, but is the wrong class	class id
XkbErr_BadId	0xfd	device found, class ok, but device does not have a feedback with the indicated id	feedback id

Side-Effects of Errors

With the exception of *Alloc* or *Implementation* errors, which might result in an inconsistent internal state, no XKB request that reports an error condition has any effect. Unless otherwise stated, requests which update some aspect of the keyboard description will not apply only part of a request — if part of a request fails, the whole thing is ignored.

Common Types

The following types are used in the request and event definitions in subsequent sections:

Name	Value
LISTofITEMs	The type LISTofITEMs is special. It is similar to the LISTofVALUE defined by the core protocol, but the elements of a LISTofITEMs are not necessarily all the same size. The use of a BITMASK to indicate which

Name	**Value**
	members are present is optional for a LISTofITEMs — it is possible for the set of elements to be derived from one or more fields of the request.
KB_DEVICESPEC	8 bit unsigned integer, *UseCoreKbd, or UseCorePtr*
KB_LEDCLASSSPEC	{ *KbdFeedbackClass* , *LedFeedbackClass* , *DfltXIClass* , *AllXIClasses* , *XINone* }
KB_BELLCLASSSPEC	{ *KbdFeedbackClass* , *BellFeedbackClass* , *DfltXIClass* , *AllXIClasses* }
KB_IDSPEC	8 bit unsigned integer or *DfltXIId*
KB_VMODMASK	CARD16, each bit corresponds to a virtual modifier
KB_GROUPMASK	{ *Group1* , *Group2* , *Group3* , *Group4* }
KB_GROUPSWRAP	{ *WrapIntoRange* , *ClampIntoRange* , *RedirectIntoRange* }
KB_GROUPINFO	{ groupsWrap: KB_GROUPSWRAP redirectGroup: 1…4, numGroups: 1…4 }
KB_NKNDETAILSMASK	{ *NKN_Keycodes* , NKN_Geometry, *NKN_DeviceID* }
KB_STATEMASK	KEYBUTMASK or KB_GROUPMASK
KB_STATEPARTMASK	{ *ModifierState* , *ModifierBase* , *ModifierLatch* , *ModifierLock* , *GroupState* , *GroupBase* , *GroupLatch* , *GroupLock* , *CompatState* , *GrabMods* , *CompatGrabMods* , *LookupMods* , *CompatLookupMods* , *PointerButtons* }
KB_BOOLCTRLMASK	{ *RepeatKeys* , *SlowKeys* , *BounceKeys* , *StickyKeys* , *MouseKeys* , *MouseKeysAccel* , *AccessXKeys* , *AccessXTimeout* , *AccessXFeedback* , *AudibleBell* , *Overlay1* , *Overlay2* , *IgnoreGroupLock* }
KB_CONTROLSMASK	{ *GroupsWrap, InternalMods* , *IgnoreLockMods* , *PerKeyRepeat* , *ControlsEnabled* } or KB_BOOLCTRLMASK
KB_MAPPARTMASK	{ *KeyTypes* , *KeySyms* , *ModifierMap* , *ExplicitComponents* , *KeyActions* , *KeyBehaviors* , *VirtualMods* , *VirtualModMap* }
KB_CMDETAILMASK	{ *SymInterp* , *GroupCompat* }
KB_NAMEDETAILMASK	{ *KeycodesName* , *GeometryName* , *SymbolsName* , *PhysSymbolsName* , *TypesName* , *CompatName* , *KeyTypeNames* , *KTLevelNames* , *IndicatorNames* , *KeyNames* , *KeyAliases* , *VirtualModNames* , *GroupNames* , *RGNames* }
KB_AXNDETAILMASK	{ *AXN_SKPress* , *AXN_SKAccept* , *AXN_SKReject* , *AXN_SKRelease, AXN_BKAccept, AXN_BKReject, AXN_AXKWarning* }
KB_AXSKOPTSMASK	{ *AX_TwoKeys* , *AX_LatchToLock* }
KB_AXFBOPTSMASK	{ *AX_SKPressFB* , *AX_SKAcceptFB* , *AX_FeatureFB* , *AX_SlowWarnFB* , *AX_IndicatorFB* , *AX_StickyKeysFB* , *AX_SKReleaseFB* ,

Name	**Value**
	AX_SKRejectFB , *AX_BKRejectFB* , *AX_DumbBellFB* }
KB_AXOPTIONSMASK	KB_AXFBOPTSMASK or KB_AXSKOPTSMASK
KB_GBNDETAILMASK	{ *GBN_Types* , *GBN_CompatMap* , *GBN_ClientSymbols* , *GBN_ServerSymbols* , *GBN_IndicatorMap* , *GBN_KeyNames* , *GBN_Geometry* , *GBN_OtherNames* }
KB_BELLDETAILMASK	{ *XkbAllBellNotifyEvents* }
KB_MSGDETAILMASK	{ *XkbAllActionMessages* }
KB_EVENTTYPE	{ *XkbNewKeyboardNotify* , *XkbMapNotify* , *XkbStateNotify* , *XkbControlsNotify* , *XkbIndicatorStateNotify* , *XkbIndicatorMapNotify* , *XkbNamesNotify* , *XkbCompatMapNotify* , *XkbBellNotify* , *XkbActionMessage* , *XkbAccessXNotify* , *XkbExtensionDeviceNotify* }
KB_ACTION	[type: CARD8 data: LISTofCARD8]
KB_BEHAVIOR	[type: CARD8, data: CARD 8]
KB_MODDEF	[mask: KEYMASK, mods: KEYMASK, vmods: KB_VMODMASK]
KB_KTMAPENTRY	[active: BOOL, level: CARD8, mods: KB_MODDEF]
KB_KTSETMAPENTRY	[level: CARD8, mods: KB_MODDEF]
KB_KEYTYPE	[mods: KB_MODDEF, numLevels: CARD8, map: LISTofKB_KTMAPENTRY, preserve: LISTofKB_MODDEF]
KB_SETKEYTYPE	[realMods: KEYMASK, vmods: CARD16, numLevels: CARD8, map: LISTofKB_KTSETMAPENTRY, preserve: LISTofKB_MODDEF]
KB_KEYSYMMAP	[ktIndex: LISTofCARD8, width: CARD8 numGroups: 0…4, groupsWrap: KB_GROUPSWRAP, redirectGroup: 0…3, syms: LISTofKEYSYM]
KB_KEYVMODMAP	[key: KEYCODE, vmods: CARD16]
KB_KEYMODMAP	[key: KEYCODE, mods: KEYMASK]
KB_EXPLICITMASK	{ *ExplicitKeyType1* , *ExplicitKeyType2* , *ExplicitKeyType3* , *ExplicitKeyType4* , *ExplicitInterpret* , *ExplicitAutoRepeat* , *ExplicitBehavior* , *ExplicitVModMap* }
KB_INDICATORMASK	CARD32, each bit corresponds to an indicator
KB_IMFLAGS	{ *IM_NoExplicit* , *IM_NoAutomatic* , *IM_LEDDrivesKB* }
KB_IMMODSWHICH	{ *IM_UseNone* , *IM_UseBase* , *IM_UseLatched* , *IM_UseLocked* , *IM_UseEffective* , *IM_UseCompat* }
KB_IMGROUPSWHICH	{ *IM_UseNone* , *IM_UseBase* , *IM_UseLatched* , *IM_UseLocked* , *IM_UseEffective* }

Name	Value
KB_INDICATORMAP	[flags: CARD8, mods: KB_MODDEF, whichMods: groups: KB_GROUPMASK, whichGroups: ctrls: KB_BOOLCTRLMASK]
KB_SYMINTERPMATCH	{ *SI_NoneOf* , *SI_AnyOfOrNone* , *SI_AnyOf* , *SI_AllOf* , *SI_Exactly* }
KB_SYMINTERP	[sym: KEYSYM, mods; KEYMASK, levelOneOnly: BOOL, match: KB_SYMINTERPMATCH, virtualMod: CARD8, autoRepeat: BOOL, lockingKey: BOOL]
KB_PCFMASK	{ *PCF_DetectableAutorepeat* , *PCF_GrabsUseXkbState* , *PCF_AutoResetControls* , *PCF_LookupStateWhenGrabbed* , *PCF_SendEventUsesXKBState* }
KB_LCFLAGSMASK	{ *LC_Hidden* , *LC_Default* , *LC_Partial* }
KB_LCSYMFLAGSMASK	{ *LC_AlphanumericKeys* , *LC_ModifierKeys* , *LC_KeypadKeys* , *LC_FunctionKeys* , *LC_AlternateGroup* }

These types are used by the *XkbGetGeometry* and *XkbSetGeometry* requests:

Name	Value
KB_PROPERTY	[name, value: STRING8]
KB_POINT	[x, y: CARD16]
KB_OUTLINE	[cornerRadius: CARD8, points: LISTofKB_POINT]
KB_SHAPE	[name: ATOM, outlines: LISTofKB_OUTLINE primaryNdx, approxNdx: CARD8]
KB_KEYNAME	[name: LISTofCHAR]
KB_KEYALIAS	[real: LISTofCHAR, alias: LISTofCHAR]
KB_KEY	[name: KB_KEYNAME, gap: INT16, shapeNdx, colorNdx: CARD8]
KB_ROW	[top, left: INT16, vertical: BOOL, keys LISTofKB_KEY]
KB_OVERLAYKEY	[over, under: KB_KEYNAME]
KB_OVERLAYROW	[rowUnder: CARD8, keys: LISTofKB_OVERLAYKEY]
KB_OVERLAY	[sectionUnder: CARD8, rows: LISTofKB_OVERLAYROW]
KB_SHAPEDOODAD	[name: ATOM, priority: CARD8, top, left: INT16, type: { SolidDoodad, OutlineDoodad }, angle: INT16, width, height: CARD16 colorNdx, shapeNdx: CARD8]
KB_TEXTDOODAD	[name: ATOM, priority: CARD8, top, left: INT16, angle: INT16, width, height: CARD16, colorNdx: CARD8, text: STRING8, font: STRING8]
KB_INDICATORDOODAD	[name: ATOM, priority: CARD8, top, left: INT16, angle: INT16, shapeNdx, onColorNdx, offColorNdx: CARD8]

Name	**Value**
KB_LOGODOODAD	[name: ATOM, priority: CARD8, top, left: INT16, angle: INT16, colorNdx, shapeNdx: CARD8, logoName: STRING8]
KB_DOODAD	KB_SHAPEDOODAD, or KB_TEXTDOODAD, or KB_INDICATORDOODAD, or KB_LOGODOODAD
KB_SECTION	[name: ATOM, top, left, angle: INT16, width, height: CARD16, priority: CARD8, rows: LISTofKB_ROW, doodads: LISTofKB_DOODAD, overlays: LISTofKB_OVERLAY]

These types are used by *XkbGetDeviceInfo* and *XkbSetDeviceInfo* :

Name	**Value**
KB_XIDEVFEATUREMASK	{ *XI_ButtonActions* , *XI_IndicatorNames* , *XI_IndicatorMaps* , *XI_IndicatorState* }
KB_XIFEATUREMASK	{ KB_XIDEVFEATURES or *XI_Keyboards*
KB_XIDETAILMASK	{ KB_XIFEATURES or *XI_UnsupportedFeature* ▯
KB_DEVICELEDINFO	[ledClass: KB_LEDCLASSSPEC, ledID: KB_IDSPEC, physIndicators: CARD32, state: CARD32, names: LISTofATOM, maps: LISTofKB_INDICATORMAP]

Requests

This section lists all of the requests supported by the X Keyboard Extension, separated into categories of related requests.

Initializing the X Keyboard Extension

XkbUseExtension

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 and successfully requests the XKB extension, an XKB capable server reports compatibility state in all core protocol events and requests. Once a client asks for XKB extension semantics by issuing this request, the server reports the extended XKB keyboard state in some core protocol events and requests, as described in the overview section of this specification.

Clients should issue an *XkbUseExtension* request before using any other extension requests.

Selecting Events

XkbSelectEvents

deviceSpec: KB_DEVICESPEC

affectWhich, clear, selectAll: KB_EVENTTYPE

affectMap, map: KB_MAPPARTMASK

details: LISTofITEMs

Errors: *Keyboard* , *Match* , *Value*

This request updates the event masks of the keyboard indicated by *deviceSpec* for this client. If *deviceSpec* specifies an illegal device, a *Keyboard* error results.

The *affectMap* and *map* fields specify changes to the event details mask for the *XkbMapNotify* event. If any map components are set in *map* but not in *affectMap* , a *Match* error results. Otherwise, any map components that are set in *affectMap* are set or cleared in the map notify details mask, depending on the value of the corresponding field in *map* .

The *affectWhich* , *clear* , and *selectAll* fields specify changes to any other event details masks. If any event types are set in both *clear* and *selectAll* , a *Match* error results; if any event types are specified in either *clear* or *selectAll* but not in *affectWhich* , a *Match* error results. Otherwise, the detail masks for any event types specified in the *affectWhich* field of this request are changed as follows:

- If the event type is also set in *clear* , the detail mask for the corresponding event is set to *0* or *False* , as appropriate.
- If the event type is also set in *selectAll* , the detail mask for the corresponding event is set to include all legal detail values for that type.
- If the event type is not set in either *clear* or *selectAll* , the corresponding element of *details* lists a set of explicit changes to the details mask for the event, as described below.

Each entry of the *details* list specifies changes to the event details mask for a single type of event, and consists of an *affects* mask and a *values* mask. All details that are specified in *affects* are set to the corresponding value from *values* ; if any details are listed in *values* but not in *affects* , a *Match* error results.

The details list contains entries only for those event types, if any, that are listed in the *affectWhich* mask and not in either *clear* or *selectAll* . When present, the items of the *details* list appear in the following order:

Event Type	**Legal Details**	**Type**
XkbNewKeyboardNotify	*KB_NKNDETAILSMASK*	*CARD16*
XkbStateNotify	*KB_STATEPARTMASK*	*CARD16*
XkbControlsNotify	*KB_CONTROLMASK*	*CARD32*
XkbIndicatorMapNotify	*KB_INDICATORMASK*	*CARD32*
XkbIndicatorStateNotify	*KB_INDICATORMASK*	*CARD32*
XkbNamesNotify	*KB_NAMEDETAILMASK*	*CARD16*

Event Type	Legal Details	Type
XkbCompatMapNotify	*KB_CMDETAILMASK*	*CARD8*
XkbBellNotify	*KB_BELLDETAILMASK*	*CARD8*
XkbActionMessage	*KB_MSGDETAILMASK*	*CARD8*
XkbAccessXNotify	*KB_AXNDETAILMASK*	*CARD16*
XkbExtensionDeviceNotify	*KB_XIDETAILMASK*	*CARD16*

Detail masks for event types that are not specified in *affectWhich* are not changed.

If any components are specified in a client's event masks, the X server sends the client an appropriate event whenever any of those components change state. Unless explicitly modified, all event detail masks are empty. Events describes all XKB events and the conditions under which the server generates them.

Generating Named Keyboard Bells

XkbBell
deviceSpec: KB_DEVICESPEC
bellClass: KB_BELLCLASSSPEC
bellID: KB_IDSPEC
percent: INT8
forceSound: BOOL
eventOnly: BOOL
pitch, duration: INT16
name: ATOM
window: WINDOW
Errors: *Keyboard* , *Value* , *Match*

This request generates audible bells and/or *XkbBellNotify* events for the bell specified by the *bellClass* and *bellID* on the device specified by *deviceSpec* at the specified *pitch* , *duration* and volume (*percent*). If deviceSpec specifies a device that does not have a bell or keyboard feedback, a *Keyboard* error results.

If both *forceSound* and *eventOnly* are set, this request yields a *Match* error. Otherwise, if *forceSound* is *True* , this request always generates a sound and never generates an event; if *eventOnly* is *True* , it causes an event but no sound. If neither *forceSound* nor *eventOnly* are *True* , this request always generates an event; if the keyboard's global *AudibleBell* control is enabled, it also generates a sound.

Any bell event generated by this request contains all of the information about the bell that was requested, including the symbolic name specified by *name* and the event window specified by window. The *name* and *window* are not directly interpreted by XKB, but they must have the value *None* or specify a legal Atom or Window, respectively. *XkbBellNotify* events generated in response to core protocol or X input extension bell requests always report *None* as their *name* .

The *bellClass* , *bellID* , and *percent* fields are interpreted as for the X input extension *DeviceBell* request. If *pitch* and *duration* are zero, the server uses the corresponding values for that bell from the core protocol or input extension, otherwise *pitch* and *duration* are interpreted as for the core protocol *ChangeKeyboardControl* request; if they do not include legal values, a *Value* error results. The *window* field must specify a legal Window or have the value *None* , or a *Value* error results. The name field must specify a legal Atom or have the value *None* , or an *Atom* error results. If an error occurs, this request has no other effect (i.e. does not cause a sound or generate an event).

The *pitch* , *volume* , and *duration* are suggested values for the bell, but XKB does not require the server to honor them.

Querying and Changing Keyboard State

XkbGetState

deviceSpec: KB_DEVICESPEC

deviceID: CARD8

mods, baseMods, latchedMods, lockedMods: KEYMASK

group, lockedGroup: KB_GROUP

baseGroup, latchedGroup: INT16

compatState: KEYMASK

grabMods, compatGrabMods: KB_GROUP

lookupMods, compatLookupMods: KEYMASK

ptrBtnState: BUTMASK

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 *XkbLatchLockState* requests. The *group* return value reports the effective keyboard group which results from the current combination of base, latched and locked group values.

The *lookupMods* return value reports the lookup modifiers, which consist of the current effective modifiers minus any server internal modifiers. The *grabMods* return value reports the grab modifiers, which consist of the lookup modifiers minus any members of the ignore locks mask that are not either latched or logically depressed. Keyboard State describes the lookup modifiers and grab modifiers in more detail.

The *ptrBtnState* return value reports the current logical state of up to five buttons on the core pointer device.

The *compatState* return value reports the compatibility state that corresponds to the effective keyboard group and modifier state. The *compatLookupMods* and *compatGrabMods* return values report the core protocol compatibility states that correspond to the XKB lookup and grab state. All of the compatibility states are computed by applying the group compatibility mapping to the corresponding XKB modifier and group states, as described in Group Compatibility Map.

XkbLatchLockState

deviceSpec: KB_DEVICESPEC

affectModLocks, modLocks: KEYMASK

lockGroup: BOOL

groupLock: KB_GROUP

affectModLatches,modLatches: KEYMASK

latchGroup: BOOL

groupLatch: INT16

Errors: *Keyboard* , *Value*

This request locks or latches keyboard modifiers and group state for the device specified by *deviceSpec* . If *deviceSpec* specifies an illegal or non-keyboard device, a *Keyboard* error occurs.

The locked state of any modifier specified in the *affectModLocks* mask is set to the corresponding value from *modLocks* . If *lockGroup* is *True* , the locked keyboard group is set to the group specified by *groupLock* . If any modifiers are set in *modLocks* but not *affectModLocks* , a *Match* error occurs.

The latched state of any modifier specified in the *affectModLatches* mask is set to the corresponding value from *modLatches* . If *latchGroup* is *True* , the latched keyboard group is set to the group specified by *groupLatch* . if any modifiers are set in *modLatches* but not in *affectModLatches* , a *Match* error occurs.

If the locked group exceeds the maximum number of groups permitted for the specified keyboard, it is wrapped or truncated back into range as specified by the global *GroupsWrap* control. 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 *XkbStateNotify* events as appropriate if any state components have changed. Changing the keyboard state might also turn indicators on or off which can cause *XkbIndicatorStateNotify* events as well.

If any errors occur, this request has no effect.

Querying and Changing Keyboard Controls

XkbGetControls
deviceSpec: KB_DEVICESPEC
deviceID: CARD8
mouseKeysDfltBtn: CARD8
numGroups: CARD8
groupsWrap: KB_GROUPINFO
internalMods,ignoreLockMods: KB_MODDEF
repeatDelay,repeatInterval: CARD16
slowKeysDelay, debounceDelay: CARD16
mouseKeysDelay, mouseKeysInterval: CARD16
mouseKeysTimeToMax, mouseKeysMaxSpeed: CARD16
mouseKeysCurve: INT16
accessXOptions: KB_AXOPTIONMASK
accessXTimeout: CARD16
accessXTimeoutOptionsMask, accessXTimeoutOptionValues: CARD16
accessXTimeoutMask,accessXTimeoutValues: CARD32
enabledControls: KB_BOOLCTRLMASK
perKeyRepeat: LISTofCARD8
Errors: *Keyboard*

This request returns the current values and status of all controls for the keyboard specified by *deviceSpec* . If *deviceSpec* specifies an illegal device a *Keyboard* error results. 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, and *groupsWrap* reports the treatment of out-of-range groups, as described in Key Symbol Map. The *internalMods* and *ignoreLockMods* return values report the current values of the server internal and ignore locks modifiers as described in Keyboard State. Both are modifier definitions (Modifier Definitions) which report the real modifiers, virtual modifiers, and the resulting combination of real modifiers that are bound to the corresponding control.

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. The *mouseKeysDelay* , *mouseKeysInterval* , *mouseKeysTimeToMax* and *mouseKeysMaxSpeed* and *mouseKeysCurve* return values report the current acceleration applied to mouse keys, as described in The MouseKeysAccel Control. All times are reported in milliseconds.

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

The *accessXOptions* return value reports the current settings of the various AccessX options flags which govern the behavior of the *StickyKeys* control and of AccessX feedback.

The *accessXTimeout* return value reports the length of time, in seconds, that the keyboard must remain idle before AccessX controls are automatically changed; an *accessXTimeout* of *0* indicates that AccessX controls are not automatically changed. The *accessXTimeoutMask* specifies the boolean controls to be changed if the AccessX timeout expires; the *accessXTimeoutValues* field specifies new values for all of the controls in the timeout mask. The *accessXTimeoutOptionsMask* field specifies the AccessX options to be changed when the AccessX timeout expires; the *accessXTimeoutOptionValues* return value reports the values to which they will be set.

The *enabledControls* return value reports the current state of all of the global boolean controls.

The *perKeyRepeat* array consists of one bit per key and reports the current autorepeat behavior of each keyboard key; if a bit is set in *perKeyRepeat* , the corresponding key repeats if it is held down while global keyboard autorepeat is enabled. This array parallels the core protocol and input extension keyboard controls, if the autorepeat behavior of a key is changed via the core protocol or input extension, those changes are automatically reflected in the *perKeyRepeat* array.

XkbSetControls

deviceSpec: KB_DEVICESPEC
affectInternalRealMods, internalRealMods: KEYMASK
affectInternalVirtualMods,internalVirtualMods: KB_VMODMASK
affectIgnoreLockRealMods,ignoreLockRealMods: KB_MODMASK
affectIgnoreLockVirtualMods,ignoreLockVirtualMods: KB_VMODMASK
mouseKeysDfltBtn: CARD8
groupsWrap: KB_GROUPINFO
accessXOptions: CARD16
affectEnabledControls: KB_BOOLCTRLMASK
enabledControls: KB_BOOLCTRLMASK
changeControls: KB_CONTROLMASK
repeatDelay,repeatInterval: CARD16
slowKeysDelay, debounceDelay: CARD16
mouseKeysDelay, mouseKeysInterval: CARD16
mouseKeysTimeToMax, mouseKeysMaxSpeed: CARD16
mouseKeysCurve: INT16
accessXTimeout: CARD16
accessXTimeoutMask, accessXTimeoutValues: KB_BOOLCTRLMASK
accessXTimeoutOptionsMask,accessXTimeoutOptionsValues: CARD16
perKeyRepeat: LISTofCARD8
Errors: *Keyboard* , *Value*

This request sets the keyboard controls indicated in *changeControls* for the keyboard specified by *deviceSpec* . Each bit that is set in *changeControls* indicates that one or more of the other request fields should be applied, as follows:

Bit in changeControls	Field(s) to be Applied
XkbRepeatKeysMask	*repeatDelay* , *repeatInterval*
XkbSlowKeysMask	*slowKeysDelay*
XkbStickyKeysMask	*accessXOptions* (only the *XkbAX_TwoKeys* and the *XkbAX_LatchToLock* options are affected)
XkbBounceKeysMask	*debounceDelay*
XkbMouseKeysMask	*mouseKeysDfltBtn*
XkbMouseKeysAccelMask	*mouseKeysDelay* , *mouseKeysInterval* , *mouseKeysCurve* , *mouseKeysTimeToMax* , *mouseKeysMaxSpeed*
XkbAccessXKeysMask	*accessXOptions (all options)*
XkbAccessXTimeoutMask	*accessXTimeout* , *accessXTimeoutMask* , *accessXTimeoutValues* , *accessXTimeoutOptionsMask* , *accessXTimeoutOptionsValues*
XkbAccessXFeedbackMask	*accessXOptions* (all options except those affected by the *XkbStickyKeysMask* bit)
XkbGroupsWrapMask	*groupsWrap*
XkbInternalModsMask	*affectInternalRealMods* , *internalRealMods* , *affectInternalVirtualMods* , *internalVirtualMods*
XkbIgnoreLockModsMask	*affectIgnoreLockRealMods* , *ignoreLockRealMods* , *affectIgnoreLockVirtualMods* , *ignoreLockVirtualMods*
XkbPerKeyRepeatMask	*perKeyRepeat*
XkbControlsEnabledMask	*affectEnabledControls* , *enabledControls*

If any other bits are set in *changeControls* , a *Value* error results. If any of the bits listed above are not set in *changeControls* , the corresponding fields must have the value *0* , or a *Match* error results.

If applied, *repeatDelay* and *repeatInterval* change the autorepeat characteristics of the keyboard, as described in The RepeatKeys Control. If specified, *repeatDelay* and *repeatInterval* must both be non-zero or a *Value* error results.

If applied, the *slowKeysDelay* field specifies a new delay for the *SlowKeys* control, as defined in The SlowKeys Control. If specified, *slowKeysDelay* must be non-zero, or a *Value* error results.

If applied, the *debounceDelay* field specifies a new delay for the *BounceKeys* control, as described in The BounceKeys Control. If present, the *debounceDelay* must be non-zero or a *Value* error results.

If applied, the *mouseKeysDfltBtn* field specifies the core pointer button for which events are generated whenever a *SA_PtrBtn* or *SA_LockPtrBtn* key action is activated. If present, *mouseKeysDfltBtn* must specify a legal button for the core pointer device, or a *Value* error results. Key Actions describes the *SA_PtrBtn* and *SA_LockPtrBtn* actions in more detail.

If applied, the *mouseKeysDelay* , *mouseKeysInterval* , *mouseKeysTimeToMax* , *mouseKeysMaxSpeed* and *mouseKeysCurve* fields change the rate at which the pointer moves when a key which generates a *SA_MovePtr* action is held down. The MouseKeysAccel Control describes these *MouseKeysAccel* parameters in more detail. If defined, the *mouseKeysDelay* , *mouseKeysInterval* , *mouseKeysTimeToMax* and *mouseKeysMaxSpeed* values must all be greater than zero, or a *Value* error results. The *mouseKeysCurve* value must be greater than *-1000* or a *Value* error results.

If applied, the *accessXOptions* field sets the AccessX options, which are described in detail in The AccessXKeys Control. If either one of *XkbStickyKeysMask* and *XkbAccessXFeedbackMask* are set in *changeControls* and *XkbAccessXKeysMask* is not, only a subset of the AccessX options are changed, as described in the table above; if both are set or if the *AccessXKeys* bit is set in *changeControls* , all of the AccessX options are updated. Any bit in *accessXOptions* whose interpretation is undefined must be zero, or a *Value* error results.

If applied, the *accessXTimeout* , *accessXTimeoutMask* , *accessXTimeoutValues* , *accessXTimeoutOptionsMask* and *accessXTimeoutOptionsValues* fields change the behavior of the AccessX Timeout control, as described in The AccessXTimeout Control. The *accessXTimeout* must be greater than zero, or a *Value* error results. The *accessXTimeoutMask* or *accessXTimeoutValues* fields must specify only legal boolean controls, or a *Value* error results. The *accessXTimeoutOptionsMask* and *accessXTimeoutOptionsValues* fields must contain only legal AccessX options or a *Value* error results. If any bits are set in either values field but not in the corresponding mask, a *Match* error results.

If present, the *groupsWrap* field specifies the treatment of out-of-range keyboard groups, as described in Key Symbol Map. If the *groupsWrap* field does not specify a legal treatment for out-of-range groups, a *Value* error results.

If present, the *affectInternalRealMods* field specifies the set of real modifiers to be changed in the internal modifier definition and the *internalRealMods* field specifies new values for those modifiers. The *affectInternalVirtualMods* and *internalVirtualMods* fields update the virtual modifier component of the modifier definition that describes the internal modifiers in the same way. If any bits are set in either values field but not in the corresponding mask field, a *Match* error results.

If present, the *affectIgnoreLockRealMods* field specifies the set of real modifiers to be changed in the ignore locks modifier definition and the *ignoreLockRealMods* field specifies new values for those modifiers. The *affectIgnoreLockVirtualMods* and *ignoreLockVirtualMods* fields update the virtual modifier component of the ignore locks modifier definition in the same way. If any bits are set in either values field but not in the corresponding mask field, a *Match* error results.

If present, the *perKeyRepeat* array specifies the repeat behavior of the individual keyboard keys. The corresponding core protocol or input extension per-key autorepeat information is updated to reflect any changes specified in *perKeyRepeat* . If the bits that correspond to any out-of-range keys are set in *perKeyRepeat* , a *Value* error results.

If present, the *affectEnabledControls* and *enabledControls* field enable and disable global boolean controls. Any controls set in both fields are enabled; any controls that are set in *affectEnabledControls* but not in *enabledControls* are disabled. Controls that are not set in either field are not affected. If any controls are specified in *enabledControls* but not in *affectEnabledControls* , a *Match* error results. If either field contains anything except boolean controls, a *Value* error results.

Querying and Changing the Keyboard Mapping

XkbGetMap
deviceSpec: KB_DEVICESPEC
full, partial: KB_MAPPARTMASK
firstType, nTypes: CARD8
firstKeySym, firstKeyAction: KEYCODE
nKeySyms, nKeyActions: CARD8
firstKeyBehavior,firstKeyExplicit: KEYCODE
nKeyBehaviors,nKeyExplicit: CARD8
firstModMapKey,firstVModMapKey: KEYCODE
nModMapKeys, nVModMapKeys: CARD8
virtualMods: KB_VMODMASK
deviceID: CARD8
minKeyCode, maxKeyCode: KEYCODE
present: KB_MAPPARTMASK
firstType, nTypes, nTotalTypes: CARD8
firstKeySym, firstKeyAction: KEYCODE
nKeySyms, nKeyActions: CARD8
totalSyms, totalActions: CARD16
firstKeyBehavior, firstKeyExplicit: KEYCODE
nKeyBehaviors, nKeyExplicit: CARD8
totalKeyBehaviors, totalKeyExplicit: CARD8
firstModMapKey, firstVModMapKey: KEYCODE
nModMapKeys, nVModMapKeys: CARD8
totalModMapKeys, totalVModMapKeys: CARD8
virtualMods: KB_VMODMASK
typesRtrn: LISTofKB_KEYTYPE
symsRtrn: LISTofKB_KEYSYMMAP
actsRtrn: { count: LISTofCARD8, acts: LISTofKB_ACTION }
behaviorsRtrn: LISTofKB_SETBEHAVIOR
vmodsRtrn: LISTofSETofKEYMASK
explicitRtrn: LISTofKB_SETEXPLICIT
modmapRtrn: LISTofKB_KEYMODMAP
vmodMapRtrn: LISTofKB_KEYVMODMAP
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* , or a *Value* error if any undefined bits are set in either *full* or *partial* .

Each bit in the *partial* mask controls the interpretation of one or more of the other request fields, as follows:

Bit in the Partial Mask	**Type**	**Corresponding Field(s)**
XkbKeyTypesMask	key types	*firstType* , *nTypes*
XkbKeySymsMask	keycodes	*firstKeySym* , *nKeySyms*
XkbKeyActionsMask	keycodes	*firstKeyAction* , *nKeyActions*
XkbKeyBehaviorsMask	keycodes	*firstKeyBehavior* , *nKeyBehaviors*
XkbExplicitComponentsMask	keycodes	*firstKeyExplicit* , *nKeyExplicit*
XkbModifierMapMask	keycodes	*firstModMapKey* , *nModMapKeys*
XkbVirtualModMapMask	keycodes	*firstVModMapKey* , *nVModMapKeys*
XkbVirtualModsMask	virtual modifiers	*virtualMods*

If any of these keyboard map components are specified in *partial* , the corresponding values must specify a valid subset of the requested components or this request reports a *Value* error. If a keyboard map component is not specified in *partial* , the corresponding fields must contain zeroes, or a *Match* error results.

If any error is generated, the request aborts and does not report any values.

On successful return, the *deviceID* field reports the X input extension device ID of the keyboard for which information is being returned, or *0* if the server does not support the X input extension. The *minKeyCode* and *maxKeyCode* return values report the minimum and maximum keycodes that are legal for the keyboard in question.

The *present* return value lists all of the keyboard map components contained in the reply. The bits in *present* affect the interpretation of the other return values as follows:

If *XkbKeyTypesMask* is set in *present* :

- *firstType* and *nTypes* specify the types reported in the reply.
- *nTotalTypes* reports the total number of types defined for the keyboard
- *typesRtrn* has *nTypes* elements of type KB_KEYTYPE which describe consecutive key types starting from *firstType* .

If *XkbKeySymsMask* is set in *present* :

- *firstKeySym* and *nKeySyms* specify the subset of the keyboard keys for which symbols will be reported.
- *totalSyms* reports the total number of keysyms bound to the keys returned in this reply.

- *symsRtrn* has *nKeySyms* elements of type KB_KEYSYMMAP, which describe the symbols bound to consecutive keys starting from *firstKeySym* .

If *XkbKeyActionsMask* is set in *present* :

- *firstKeyAction* and *nKeyActions* specify the subset of the keys for which actions are reported.

- *totalActions* reports the total number of actions bound to the returned keys.

- The *count* field of the *actsRtrn* return value has *nKeyActions* entries of type CARD8, which specify the number of actions bound to consecutive keys starting from *firstKeyAction* . The *acts* field of *actsRtrn* has *totalActions* elements of type KB_ACTION and specifies the actions bound to the keys.

If *XkbKeyBehaviorsMask* is set in *present* :

- The *firstKeyBehavior* and *nKeyBehaviors* return values report the range of keyboard keys for which behaviors will be reported.

- The *totalKeyBehaviors* return value reports the number of keys in the range to be reported that have non-default values.

- The *behaviorsRtrn* value has *totalKeyBehaviors* entries of type KB_BEHAVIOR. Each entry specifies a key in the range for which behaviors are being reported and the behavior associated with that key. Any keys in that range that do not have an entry in *behaviorsRtrn* have the default behavior, *KB_Default* .

If *XkbExplicitComponentsMask* is set in *present* :

- The *firstKeyExplicit* and *nKeyExplicit* return values report the range of keyboard keys for which the set of explicit components is to be returned.

- The *totalKeyExplicit* return value reports the number of keys in the range specified by *firstKeyExplicit* and *nKeyExplicit* that have one or more explicit components.

- The *explicitRtrn* return value has *totalKeyExplicit* entries of type KB_KEYEXPLICIT. Each entry specifies the a key in the range for which explicit components are being reported and the explicit components that are bound to it. Any keys in that range that do not have an entry in *explicitRtrn* have no explicit components.

If *XkbModifierMapMask* is set in *present* :

- The *firstModMapKey* and *nModMapKeys* return values report the range of keyboard keys for which the modifier map is to be reported.

- The *totalModMapKeys* return value reports the number of keys in the range specified by *firstModMapKey* and *nModMapKeys* that are bound with to one or more modifiers.

- The *modmapRtrn* return value has *totalModMapKeys* entries of type KB_KEYMODMAP. Each entry specifies the a key in the range for which the modifier map is being reported and the set of modifiers that are bound to that key. Any keys in that range that do not have an entry in *modmapRtrn* are not associated with any modifiers by the modifier mapping.

If *XkbVirtualModMapMask* is set in *present* :

- The *firstVModMapKey* and *nVModMapKeys* return values report the range of keyboard keys for which the virtual modifier map is to be reported.
- The *totalVModMapKeys* return value reports the number of keys in the range specified by *firstVModMapKey* and *nVModMapKeys* that are bound with to or more virtual modifiers.
- The *vmodmapRtrn* return value has *totalVModMapKeys* entries of type KB_KEYVMODMAP. Each entry specifies the a key in the range for which the virtual modifier map is being reported and the set of virtual modifiers that are bound to that key. Any keys in that range that do not have an entry in *vmodmapRtrn* are not associated with any virtual modifiers,

If *XkbVirtualModsMask* is set in *present* :

- The *virtualMods* return value is a mask with one bit per virtual modifier which specifies the virtual modifiers for which a set of corresponding real modifiers is to be returned.
- The *vmodsRtrn* return value is a list with one entry of type KEYBUTMASK for each virtual modifier that is specified in *virtualMods* . The entries in *vmodsRtrn* contain the real modifier bindings for the specified virtual modifiers, beginning with the lowest-numbered virtual modifier that is present in *virtualMods* and proceeding to the highest.

If any of these bits are not set in *present* , the corresponding numeric fields all have the value zero, and the corresponding lists are all of length zero.

XkbSetMap

deviceSpec: KB_DEVICESPEC

flags: { *SetMapResizeTypes, SetMapRecomputeActions* }

present: KB_MAPPARTMASK

minKeyCode, maxKeyCode: KEYCODE

firstType, nTypes: CARD8

firstKeySym, firstKeyAction: KEYCODE

nKeySyms, nKeyActions: CARD8

totalSyms, totalActions: CARD16

firstKeyBehavior, firstKeyExplicit: KEYCODE

nKeyBehaviors, nKeyExplicit: CARD8

totalKeyBehaviors, totalKeyExplicit: CARD8

firstModMapKey, firstVModMapKey: KEYCODE

nModMapKeys, nVModMapKeys: CARD8

totalModMapKeys, totalVModMapKeys: CARD8

virtualMods: VMODMASK

types: LISTofKB_KEYTYPE

syms: LISTofKB_KEYSYMMAP

actions: { count: LISTofCARD8, actions: LISTofKB_ACTION }

behaviors: LISTofKB_BEHAVIOR

vmods: LISTofKEYMASK

explicit: LISTofKB_EXPLICIT

modmap: LISTofKB_KEYMODMAP

vmodmap: LISTofKB_KEYVMODMAP

Errors: *Keyboard* , *Value* , *Match* , *Alloc*

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

The *present* field specifies the keyboard map components contained to be changed. The bits in *present* affect the interpretation of the other fields as follows:

If *XkbKeyTypesMask* is set in *present* , *firstType* and *nTypes* specify a subset of the key types bound to the keyboard to be changed or created. The index of the first key type to be changed must be less than or equal to the unmodified length of the list of key types or a *Value* error results.

If *XkbKeyTypesMask* is set in *present* and *SetMapResizeTypes* is set in *flags* , the server resizes the list of key types bound to the keyboard so that the last key type specified by this request is the last element in the list. If the list of key types is shrunk, any existing key definitions that use key types that eliminated are automatically assigned key types from the list of canonical key types as described in Assigning Types To Groups of Symbols for a Key. The list of key types bound to a keyboard must always include the four canonical types and cannot have more than *XkbMaxTypesPerKey* (32) types; any attempt to reduce the number of types bound to a keyboard below four or above *XkbMaxTypesPerKey* causes a *Value* error. Symbolic names for newly created key types or levels within a key type are initialized to *None* .

If *XkbKeyTypesMask* is set in *present* , the types list has *nTypes* entries of type KB_KEYTYPE.Each key type specified in *types* must be valid or a *Value* error results. To be valid a key type definition must meet the following criteria:

- The *numLevels* for the type must be greater than zero.
- If the key type is *ONE_LEVEL* (i.e. index zero in the list of key types), *numLevels* must be one.
- If the key type is *TWO_LEVEL* or *KEYPAD* , or *ALPHABETIC* (i.e. index one, two, or three in the lest of key types) group width must be two.

Each key type in types must also be internally consistent, or a Match error results. To be internally consistent, a key type definition must meet the following criteria:

- Each map entry must specify a resulting level that is legal for the type.
- Any real or virtual modifiers specified in any of the map entries must also be specified in the *mods* for the type.

If *XkbKeySymsMask* is set in *present* , *firstKeySym* and *nKeySyms* specify a subset of the keyboard keys to which new symbols are to be assigned and *totalSyms* specifies the total number of symbols to be assigned to those keys. If any of the keys specified by *firstKeySym* and *nKeySyms* are not legal, a *Match* error results. The *syms* list has *nKeySyms* elements of type KB_KEYSYMMAP. Each key in the resulting key symbol map must be valid and internally consistent or a *Value* error results. To be valid and internally consistent, a key symbol map must meet the following criteria:

- The key type indices must specify legal result key types.
- The number of groups specified by *groupInfo* must be in the range *0…4* .
- The *width* of the key symbol map must be equal to *numLevels* of the widest key type bound to the key.
- The number of symbols, *nSyms* , must equal the number of groups times *width* .

If *XkbKeyActionsMask* is set in *present* , *firstKeyAction* and *nKeyActions* specify a subset of the keyboard keys to which new actions are to be assigned and *totalActions* specifies the total number of actions to be assigned to those keys. If any of the keys specified by *firstKeyAction* and *nKeyActions* are not legal, a *Match* error results. The *count* field of the *actions* return value has *nKeyActions* elements of type CARD8; each element of *count* specifies the number of actions bound to the corresponding key. The *actions* list in the *actions* field has *totalActions* elements of type KB_ACTION. These actions are assigned to each target key in turn, as specified by *count* . The list of actions assigned to each key must either be empty or have exactly as many actions as the key has symbols, or a *Match* error results.

If *XkbKeyBehaviorsMask* is set in *present* , *firstKeyBehavior* and *nKeyBehaviors* specify a subset of the keyboard keys to which new behaviors are to be assigned, and *totalKeyBehaviors* specifies the total number of keys in that range to be assigned non-default behavior. If any of the keys specified by *firstKeyBehavior* and *nKeyBehaviors* are not legal, a *Match* error results. The *behaviors* list has *totalKeyBehaviors* elements of type KB_BEHAVIOR; each entry of *behaviors* specifies a key in the specified range and a new behavior for that key; any key that falls in the range specified by *firstBehavior* and *nBehaviors* for which no behavior is specified in *behaviors* is assigned the default behavior, *KB_Default* . The new behaviors must be legal, or a *Value* error results. To be legal, the behavior specified in the *XkbSetMap* request must:

- Specify a key in the range indicated by *firstKeyBehavior* and *nKeyBehaviors* .
- Not specify the *permanent* flag; permanent behaviors cannot be set or changed using the *XkbSetMap* request.
- If present, the *KB_Overlay1* and *KB_Overlay2* behaviors must specify a keycode for the overlay key that is valid for the current keyboard.
- If present, the *KB_RadioGroup* behavior must specify a legal index (0…31) for the radio group to which the key belongs.

Key behaviors that are not recognized by the server are accepted but ignored. Attempts to replace a "permanent" behavior are silently ignored; the behavior is not replaced, but not error is generated and any other components specified in the *XkbSetMap* request are updated, as appropriate.

If *XkbVirtualModsMask* is set in *present* , *virtualMods* is a mask which specifies the virtual modifiers to be rebound. The *vmods* list specifies the real modifiers that are bound to each of the virtual modifiers specified in *virtualMods* , starting from the lowest numbered virtual modifier and progressing upward. Any virtual modifier that is not specified in *virtualMods* has no corresponding entry in *vmods* , so the *vmods* list has one entry for each bit that is set in *virtualMods* .

If *XkbExplicitComponentsMask* is set in *present* , *firstKeyExplicit* and *nKeyExplicit* specify a subset of the keyboard keys to which new explicit components are to be assigned, and *totalKeyExplicit* specifies the total number of keys in that range that have at least one explicit component. The *explicit* list has *totalKeyExplicit* elements of type KB_KEYEXPLICIT; each entry of *explicit* specifies a key in the specified range and a new set of explicit components for that key. Any key that falls in the range specified by *firstKeyExplicit* and *nKeyExplicit* that is not assigned some value in *explicit* has no explicit components.

If *XkbModifierMapMask* is set in *present* , *firstModMapKey* and *nModMapKeys* specify a subset of the keyboard keys for which new modifier mappings are to be assigned, and *totalModMapKeys* specifies the total number of keys in that range to which at least one modifier is bound. The *modmap* list has *totalModMapKeys* elements of type KB_KEYMODMAP; each entry of *modmap* specifies a key in the specified range and a new set of modifiers to be associated with that key. Any key that falls in the range specified by *firstModMapKey* and *nModMapKeys* that is not assigned some value in *modmap* has no associated modifiers.

If the modifier map is changed by the *XkbSetMap* request, any changes are also reflected in the core protocol modifier mapping. Changes to the core protocol modifier mapping are reported to XKB-unaware clients via *MappingNotify* events and can be retrieved with the core protocol *GetModifierMapping* request.

If *XkbVirtualModMapMask* is set in *present* , *firstVModMapKey* and *nVModMapKeys* specify a subset of the keyboard keys for which new modifier mappings are to be assigned, and *totalVModMapKeys* specifies the total number of keys in that range to which at least one virtual modifier is bound. The *vmodmap* list has *totalVModMapKeys* elements of type KB_KEYVMODMAP; each entry of *vmodmap* specifies a key in the specified range and a new set of virtual modifiers to be associated with that key. Any key that falls in the range specified by *firstVModMapKey* and *nVModMapKeys* that is not assigned some value in *vmodmap* has no associated virtual modifiers.

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 *XkbSetMapRecomputeActions* bit is set in *flags* , the actions associated with any keys for which symbol or modifier bindings were changed by this request are recomputed as described in Assigning Actions To Keys. Note that actions are recomputed *after* any actions specified in this request are bound to keys, so the actions specified in this request might be clobbered by the automatic assignment of actions to keys.

If the group width of an existing key type is changed, the list of symbols associated with any keys of the changed type might be 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 calculated by applying any keyboard symbol interpretations to the corresponding symbols.

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.

Assigning key behavior *KB_RadioGroup* to a key adds that key as a member of the specified radio group. Changing a key with the existing behavior *KB_RadioGroup* removes that key from the group. Changing the elements of a radio group can cause synthetic key press or key release events if the key to be added or removed is logically down at the time of the change.

Changing a key with behavior *KB_Lock* causes a synthetic key release event if the key is logically but not physically down at the time of the change.

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

Key press and key release events caused by changing key behavior may cause additional *XkbStateNotify* or *XkbIndicatorStateNotify* events.

Querying and Changing the Compatibility Map

XkbGetCompatMap

deviceSpec: KB_DEVICESPEC
groups: KB_GROUPMASK
getAllSI: BOOL
firstSI, nSI: CARD16
deviceID: CARD8
groupsRtrn: KB_GROUPMASK
firstSIRtrn, nSIRtrn, nTotalSI: CARD16
siRtrn: LISTofKB_SYMINTERP
groupRtrn: LISTofKB_MODDEF
Errors: *Keyboard* , *Match* , *Alloc*

This request returns the listed compatibility map components for the keyboard specified by *deviceSpec* . If *deviceSpec* does not specify a valid keyboard device, a *Keyboard* Error results. On return, *deviceID* reports the input extension identifier of the keyboard device or *0* if the server does not support the input extension.

If *getAllSI* is *False* , *firstSI* and *nSI* specify a subset of the symbol interpretations to be returned; if used, *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 *getAllSyms* is *True* , the server ignores *firstSym* and *nSyms* and returns all of the symbol interpretations defined for the keyboard.

The *groups* mask specifies the groups for which compatibility maps are to be returned.

The *nTotalSI* return value reports the total number of symbol interpretations defined for the keyboard. On successful return, the *siRtrn* return list contains the definitions for *nSIRtrn* symbol interpretations beginning at *firstSIRtrn* .

The *groupRtrn* return values report the entries in the group compatibility map for any groups specified in the *groupsRtrn* return value.

XkbSetCompatMap

deviceSpec: KB_DEVICESPEC
recomputeActions: BOOL
truncateSI: BOOL
groups: KB_GROUPMASK
firstSI, nSI: CARD16
si: LISTofKB_SYMINTERPRET
groupMaps: LISTofKB_MODDEF
Errors: *Keyboard* , *Match* , *Value* , *Alloc*

This request changes a specified subset of the compatibility map of the keyboard indicated by *deviceSpec* . If *deviceSpec* specifies an invalid device, a *Keyboard* error results and nothing is changed.

The *firstSI* and *nSI* fields specify a subset of the keyboard symbol interpretations to be changed. The *si* list specifies new values for each of the interpretations in that range.

The first symbol interpretation to be changed, *firstSI* , must be less than or equal to the unchanged length of the list of symbol interpretations, or a *Value* error results. If the resulting list would be larger than the unchanged list, it server list of symbol interpretations is automatically increased in size. Otherwise, if *truncateSyms* is *True* , the server deletes any symbol interpretations after the last element changed by this request, and reduces the length of the list accordingly.

The *groupMaps* fields contain new definitions for a subset of the group compatibility map; *groups* specifies the group compatibility map entries to be updated from *groupMaps* .

All changed compatibility maps and symbol interpretations must either ignore group state or specify a legal range of groups, or a *Value* error results.

If the *recomputeActions* field is *True* , the server regenerates recalculates the actions bound to all keyboard keys by applying the new symbol interpretations to the entire key symbol map, as described in Assigning Actions To Keys.

Querying and Changing Indicators

XkbGetIndicatorState

deviceSpec: KB_DEVICESPEC

deviceID: CARD8 state: KB_INDICATORMASK

Errors: *Keyboard*

This request reports the current state of the indicators for the keyboard specified by *deviceSpec* . If *deviceSpec* does not specify a valid keyboard, a *Keyboard* error results.

On 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 *state* return value reports the state of each of the thirty-two indicators on the specified keyboard. The least-significant bit corresponds to indicator 0, the most significant bit to indicator 31; if a bit is set, the corresponding indicator is lit.

XkbGetIndicatorMap

deviceSpec: KB_DEVICESPEC

which: KB_INDICATORMASK

deviceID: CARD8

which: KB_INDICATORMASK

realIndicators: KB_INDICATORMASK

nIndicators: CARD8

maps: LISTofKB_INDICATORMAP

Errors: *Keyboard* , *Value*

This request returns a subset of the maps for the indicators on the keyboard specified by *deviceSpec* . If *deviceSpec* does not specify a valid keyboard device, a *Keyboard* error results.

The *which* field specifies the subset to be returned; a set bit in the which field indicates that the map for the corresponding indicator should be returned.

On successful return, the *deviceID* field reports the input extension identifier of the keyboard or *0* if the server does not support the input extension. Any indicators specified in *realIndicators* are 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 *XkbIndicatorStateNotify* events.

The *maps* return value reports the requested indicator maps. Indicator maps are described in Indicator Maps

XkbSetIndicatorMap

deviceSpec: KB_DEVICESPEC

which: KB_INDICATORMASK

maps: LISTofKB_INDICATORMAP

Errors: *Keyboard* , *Value*

This request changes a subset of the maps on the keyboard specified by *deviceSpec* . If *deviceSpec* does not specify a valid keyboard device, a *Keyboard* error results.

The *which* field specifies the subset to be changed; the *maps* field contains the new definitions.

If successful, the new indicator maps are applied immediately. If any indicators change state as a result of the new maps, the server generates *XkbIndicatorStateNotify* events as appropriate.

XkbGetNamedIndicator

deviceSpec: KB_DEVICESPEC

ledClass: KB_LEDCLASSSPEC

ledID: KB_IDSPEC

indicator: ATOM

deviceID: CARD8

supported: BOOL

indicator: ATOM

found: BOOL

on: BOOL

realIndicator: BOOL

ndx: CARD8

map: KB_INDICATORMAP

Errors: *Keyboard* , *Atom* , *Value*

This request returns information about the indicator specified by *ledClass* , *ledID* , and *indicator* on the keyboard specified by *deviceSpec* . The *indicator* field specifies the name of the indicator for which information is to be returned.

If *deviceSpec* does not specify a device with indicators, a *Keyboard* error results. If *ledClass* does not have the value *DfltXIClass* , *LedFeedbackClass* , or *KbdFeedbackClass* , a *Value* error results. If *ledID* does not have the value *DfltXIId* or specify the identifier of a feedback of the class specified by *ledClass* on the device specified by *deviceSpec* , a *Match* error results. If *indicator* is not a valid ATOM other than *None* , an *Atom* error results.

This request is always supported with default class and identifier on the core keyboard device. If the request specifies a device other than the core keyboard device or a feedback class and identifier other than the defaults, and the server does not support indicator names or indicator maps for extension devices, the *supported* return value is *False* and the values of the other fields in the reply are undefined. If the client which issued the unsupported request has also selected to do so, it will also receive an *XkbExtensionDeviceNotify* event which reports the attempt to use an unsupported feature, in this case one or both of *XkbXI_IndicatorMaps* or *XkbXI_IndicatorNames* .

Otherwise, *supported* is *True* and the *deviceID* field reports the input extension identifier of the keyboard or *0* if the server does not support the input extension. The *indicator* return value reports the name for which information was requested and the *found* return value is *True* if an indicator with the specified name was found on the device.

If a matching indicator was found:

- The *on* return value reports the state of the indicator at the time of the request.
- The *realIndicator* return value is *True* if the requested indicator is actually present on the keyboard or *False* if it is virtual.
- The *ndx* return value reports the index of the indicator in the requested feedback.
- The *map* return value reports the indicator map used by to automatically change the state of the specified indicator in response to changes in keyboard state or controls.

If no matching indicator is found, the *found* return value is *False* , and the *on* , *realIndicator* , *ndx* , and *map* return values are undefined.

XkbSetNamedIndicator

deviceSpec: KB_DEVICESPEC

ledClass: KB_LEDCLASSSPEC

ledID: KB_IDSPEC

indicator: ATOM

setState: BOOL

on: BOOL

setMap: BOOL

createMap: BOOL

map: KB_SETINDICATORMAP

Errors: *Keyboard* , *Atom* , *Access*

This request changes various aspects of the indicator specified by *ledClass* , *ledID* , and *indicator* on the keyboard specified by *deviceSpec* . The *indicator* argument specifies the name of the indicator to be updated.

If *deviceSpec* does not specify a device with indicators, a *Keyboard* error results. If *ledClass* does not have the value *DfltXIClass* , *LedFeedbackClass* , or *KbdFeedbackClass* , a *Value* error results. If *ledID* does not have the value *DfltXIId* or specify the identifier of a feedback of the class specified by *ledClass* on the device specified by *deviceSpec* , a *Match* error results. If *indicator* is not a valid ATOM other than *None* , an *Atom* error results.

This request is always supported with default class and identifier on the core keyboard device. If the request specifies a device other than the core keyboard device or a feedback class and identifier other than the defaults, and the server does not support indicator names or indicator maps for extension devices, the *supported* return value is *False* and the values of the other fields in the reply are undefined. If the client which issued the unsupported request has also selected to do so, it will also receive an *XkbExtensionDeviceNotify* event which reports the attempt to use an unsupported feature, in this case one or both of *XkbXI_IndicatorMaps* and *XkbXI_IndicatorNames* .

Otherwise, *supported* is *True* and the *deviceID* field reports the input extension identifier of the keyboard or *0* if the server does not support the input extension. The *indicator* return value reports the name for which information was requested and the *found* return value is *True* if an indicator with the specified name was found on the device.

If no indicator with the specified name is found on the specified device, and the *createMap* field is *True* , XKB assigns the specified name to the lowest-numbered indicator that has no name (i.e. whose name is *None*) and applies the rest of the fields in the request to the newly named indicator. If no unnamed indicators remain, this request reports no error and has no effect.

If no matching indicator is found or new indicator assigned this request reports no error and has no effect. Otherwise, it updates the indicator as follows:

If *setMap* is *True* , XKB changes the map for the indicator (see Indicator Maps) to reflect the values specified in *map* .

If *setState* is *True* , XKB attempts to explicitly change the state of the indicator to the state specified in *on* . The effects of an attempt to explicitly change the state of an indicator depend on the values in the map for that indicator and are not guaranteed to succeed.

If this request affects both indicator map and state, it updates the indicator map before attempting to change its state, so the success of the explicit change depends on the indicator map values specified in the request.

If this request changes the indicator map, it applies the new map immediately to determine the appropriate state for the indicator given the new indicator map and the current state of the keyboard.

Querying and Changing Symbolic Names

XkbGetNames

deviceSpec: KB_DEVICESPEC

which: KB_NAMEDETAILMASK

deviceID: CARD8 which: KB_NAMESMASK minKeyCode, maxKeyCode: KEYCODE nTypes: CARD8 nKTLevels: CARD16 groupNames: KB_GROUPMASK virtualMods: KB_VMODMASK firstKey: KEYCODE nKeys: CARD8 indicators: KB_INDICATORMASK nRadioGroups, nKeyAliases: CARD8 present: KB_NAMEDETAILMASK valueList: LISTofITEMs

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. If *deviceSpec* does not specify a valid keyboard device, a *Keyboard* error results. If any undefined bits in *which* are non-zero, a *Value* error results.

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 *present* and *valueList* return values specify the components for which names are being reported. If a component is specified in *present* , the corresponding element is present in the *valueList* , otherwise that component has length *0* . The components of the *valueList* appear in the following order, when present:.

Component	**Size**	**Type**
XkbKeycodesName	1	ATOM
XkbGeometryName	1	ATOM
XkbSymbolsName	1	ATOM
XkbPhysSymbolsName	1	ATOM
XkbTypesName	1	ATOM
XkbCompatName	1	ATOM
XkbKeyTypeNames	*nTypes*	LISTofATOM
XkbKTLevelNames	*nTypes* , *nKTLevels*	{ count: LISTofCARD8, names: LISTofATOM }
XkbIndicatorNames	One per bit set in *indicators*	LISTofATOM
XkbVirtualModNames	One per bit set in *virtualMods*	LISTofATOM
XkbGroupNames	One per bit set in *groupNames*	LISTofATOM
XkbKeyNames	*nKeys*	LISTofKB_KEYNAME
XkbKeyAliases	*nKeyAliases*	LISTofKB_KEYALIAS
XkbRGNames	*nRadioGroups*	LISTofATOM

If type names are reported, the *nTypes* return value reports the number of types defined for the keyboard, and the list of key type names in *valueList* has *nTypes* elements.

If key type level names are reported, the list of key type level names in the *valueList* has two parts: The *count* array has *nTypes* elements, each of which reports the number of level names reported for the corresponding key type. The *names* array has *nKTLevels* atoms and reports the names of each type sequentially. The *nKTLevels* return value is always equal to the sum of all of the elements of the *count* array.

If indicator names are reported, the *indicators* mask specifies the indicators for which names are defined; any indicators not specified in *indicators* have the name *None* . The list of indicator names in *valueList* contains the names of the listed indicators, beginning with the lowest-numbered indicator for which a name is defined and proceeding to the highest.

If virtual modifier names are reported, the *virtualMods* mask specifies the virtual modifiers for which names are defined; any virtual modifiers not specified in *virtualMods* have the name *None* . The list of virtual modifier names in *valueList* contains the names of the listed virtual modifiers, beginning with the lowest-numbered virtual modifier for which a name is defined and proceeding to the highest.

If group names are reported, the *groupNames* mask specifies the groups for which names are defined; any groups not specified in *groupNames* have the name *None* . The list of group names in *valueList* contains the names of the listed groups, beginning with the lowest-numbered group for which a name is defined and proceeding to the highest.

If key names are reported, the *firstKey* and *nKeys* return values specify a range of keys which includes all keys for which names are defined; any key that does not fall in the range specified by *firstKey* and *nKeys* has the name *NullKeyName* . The list of key names in the *valueList* has *nKeys* entries and specifies the names of the keys beginning at *firstKey* .

If key aliases are reported, the *nKeyAliases* return value specifies the total number of key aliases defined for the keyboard. The list of key aliases in *valueList* has *nKeyAliases* entries, each of which reports an alias and the real name of the key to which it corresponds.

If radio group names are reported, the *nRadioGroups* return value specifies the number of radio groups on the keyboard for which names are defined. The list of radio group names in *valueList* reports the names of each group and has *nRadioGroups* entries.

XkbSetNames

deviceSpec: KB_DEVICESPEC
which: KB_NAMEDETAILMASK
virtualMods: KB_VMODMASK
firstType, nTypes: CARD8
firstKTLevel, nKTLevels: CARD8
totalKTLevelNames: CARD16
indicators: KB_INDICATORMASK
groupNames: KB_GROUPMASK
nRadioGroups: CARD8
firstKey: KEYCODE
nKeys, nKeyAliases: CARD8
valueList: LISTofITEMs
Errors: *Keyboard* , *Atom* , *Value* , *Match* , *Alloc*

This request changes the symbolic names for the requested components of the keyboard specified by *deviceSpec* . The *which* field specifies the components for which one or more names are to be updated. If *deviceSpec* does not specify a valid keyboard device, a *Keyboard* error results. If any undefined bits in *which* are non-zero, a *Value* error results. If any error (other than *Alloc* or *Implementation*) occurs, this request returns without modifying any names.

The *which* and *valueList* fields specify the components to be changed; the type of each *valueList* entry, the order in which components appear in the *valueList* when specified, and the correspondence between components in *which* and the entries in the *valueList* are as specified for the *XkbGetNames* request.

If keycodes, geometry, symbols, physical symbols, types or compatibility map names are to be changed, the corresponding entries in the *valueList* must have the value *None* or specify a valid ATOM, else an *Atom* error occurs.

If key type names are to be changed, the *firstType* and *nTypes* fields specify a range of types for which new names are supplied, and the list of key type names in *valueList* has *nTypes* elements. Names for types that fall outside of the range specified by *firstType* and *nTypes* are not affected. If this request specifies names for types that are not present on the keyboard, a *Match* error results. All of the type names in the *valueList* must be valid ATOMs or have the value *None* , or an *Atom* error results.

The names of the first four keyboard types are specified by the XKB extension and cannot be changed; including any of the canonical types in this request causes an *Access* error, as does trying to assign the name reserved for a canonical type to one of the other key types.

If key type level names are to be changed, the *firstKTLevel* and *nKTLevels* fields specify a range of key types for which new level names are supplied, and the list of key type level names in the *valueList* has two parts: The *count* array has *nKTLevels* elements, each of which specifies the number of levels for which names are supplied on the corresponding key type; any levels for which no names are specified are assigned the name *None* . The *names* array has *totalKTLevels* atoms and specifies the names of each type sequentially. The *totalKTLevels* field must always equal the sum of all of the elements of the *count* array. Level names for types that fall outside of the specified range are not affected. If this request specifies level names for types that are not present on the keyboard, or if it specifies more names for a type than the type has levels, a *Match* error results. All specified type level names must be *None* or a valid ATOM or an *Atom* error results.

If indicator names are to be changed, the *indicators* mask specifies the indicators for which new names are specified; the names for indicators not specified in *indicators* are not affected. The list of indicator names in *valueList* contains the new names for the listed indicators, beginning with the lowest-numbered indicator for which a name is defined and proceeding to the highest. All specified indicator names must be a valid ATOM or *None* , or an *Atom* error results.

If virtual modifier names are to be changed, the *virtualMods* mask specifies the virtual modifiers for which new names are specified; names for any virtual modifiers not specified in *virtualMods* are not affected. The list of virtual modifier names in *valueList* contains the new names for the specified virtual modifiers, beginning with the lowest-numbered virtual modifier for which a name is defined and proceeding to the highest. All virtual modifier names must be valid ATOMs or *None* , or an *Atom* error results.

If group names are to be changed, the *groupNames* mask specifies the groups for which new names are specified; the name of any group not specified in *groupNames* is not changed. The list of group names in *valueList* contains the new names for the listed groups, beginning with the lowest-numbered group for which a name is defined and proceeding to the highest. All specified group names must be a valid ATOM or *None* , or an *Atom* error results.

If key names are to be changed, the *firstKey* and *nKeys* fields specify a range of keys for which new names are defined; the name of any key that does not fall in the range specified by *firstKey* and *nKeys* is not changed. The list of key names in the *valueList* has *nKeys* entries and specifies the names of the keys beginning at *firstKey* .

If key aliases are to be changed, the *nKeyAliases* field specifies the length of a new list of key aliases for the keyboard. The list of key aliases can only be replaced in its entirety; it cannot be replaced. The list of key aliases in *valueList* has *nKeyAliases* entries, each of which reports an alias and the real name of the key to which it corresponds.

XKB does not check key names or aliases for consistency and validity, so applications should take care not to assign duplicate names or aliases

If radio group names are to be changed, the *nRadioGroups* field specifies the length of a new list of radio group names for the keyboard. There is no way to edit the list of radio group names; it can only be replaced in its entirety. The list of radio group names in *valueList* reports the names of each group and has *nRadioGroups* entries. If the list of radio group names specifies names for more radio groups than XKB allows (32), a *Match* error results. All specified radio group names must be valid ATOMs or have the value *None* , or an *Atom* error results.

Querying and Changing Keyboard Geometry

XkbGetGeometry
deviceSpec: KB_DEVICESPEC
name: ATOM
deviceID: CARD8
name: ATOM
found: BOOL
widthMM, heightMM: CARD16
baseColorNdx, labelColorNdx: CARD8
properties: LISTofKB_PROPERTY
colors: LISTofSTRING8
shapes: LISTofKB_SHAPE
sections: LISTofKB_SECTION
doodads: LISTofKB_DOODAD
keyAliases: LISTofKB_KEYALIAS
Errors: *Keyboard*

This request returns a description of the physical layout of a keyboard. If the *name* field has the value *None* , or if name is identical to the name of the geometry for the keyboard specified by *deviceSpec* , this request returns the geometry of the keyboard specified by *deviceSpec* ; otherwise, if *name* is a valid atom other than *None* , the server returns the keyboard geometry description with that name in the server database of keyboard components (see The Server Database of Keyboard Components) if one exists. If *deviceSpec* does not specify a valid keyboard device, a *Keyboard* error results. If *name* has a value other than *None* or a valid ATOM, an *Atom* error results.

On successful return, the *deviceID* field reports the X Input extension identifier of the keyboard device specified in the request, or *0* if the server does not support the input extension.

The *found* return value reports whether the requested geometry was available. If *found* is *False* , no matching geometry was found and the remaining fields in the request reply are undefined; if *found* is *True* , the remaining fields of the reply describe the requested keyboard geometry. The interpretation of the components that make up a keyboard geometry is described in detail in Keyboard Geometry

XkbSetGeometry

deviceSpec: KB_DEVICESPEC
name: ATOM
widthMM, heightMM, CARD16
baseColorNdx, labelColorNdx: CARD8
shapes: LISTofKB_SHAPE
sections: LISTofKB_SECTION
properties: LISTofKB_PROPERTY
colors: LISTofSTRING8
doodads: LISTofKB_DOODAD
keyAliases: LISTofKB_KEYALIAS
Errors: *Keyboard* , *Atom* , *Value*

This request changes the reported description of the geometry for the keyboard specified by *deviceSpec* . If deviceSpec does not specify a valid keyboard device, a *Keyboard* error results.

The *name* field specifies the name of the new keyboard geometry and must be a valid ATOM or an *Atom* error results. The new geometry is not added to the server database of keyboard components, but it can be retrieved using the *XkbGetGeometry* request for as long as it is bound to the keyboard. The keyboard geometry symbolic name is also updated from the name field, and an *XkbNamesNotify* event is generated, if necessary.

The list of *colors* must include at least two definitions, or a *Value* error results. All color definitions in the geometry must specify a legal color (i.e. must specify a valid index for one of the entries of the *colors* list) or a *Match* error results. The *baseColorNdx* and the *labelColorNdx* must be different or a *Match* error results.

The list of *shapes* must include at least one shape definition, or a *Value* error results. If any two shapes have the same name, a *Match* error result. All doodads and keys which specify shape must specify a valid index for one of the elements of the *shapes* list, or a *Match* error results.

All section, shape and doodad names must be valid ATOMs or an *Atom* error results; the constant *None* is not permitted for any of these components.

All doodads must be of a known type; XKB does not support "private" doodad types.

If, after rotation, any keys or doodads fall outside of the bounding box for a section, the bounding box is automatically adjusted to the minimum size which encloses all of its components.

If, after adjustment and rotation, the bounding box of any section or doodad extends below zero on either the X or Y axes, the entire geometry is translated so that the minimum extent along either axis is zero.

If, after rotation and translation, any keyboard components fall outside of the rectangle specified by *widthMM* and *heightMM* , the keyboard dimensions are automatically resized to the minimum bounding box that surrounds all components. Otherwise, the width and height of the keyboard are left as specified.

The *under* field of any overlay key definitions must specify a key that is in the section that contains the overlay key, or a *Match* error results. This request does not check the value of the *over* field of an overlay key definition, so applications must be careful to avoid conflicts with actual keys.

This request does not verify that key names or aliases are unique. It also does not verify that all key names specified in the geometry are bound to some keycode or that all keys that are named in the keyboard definition are also available in the geometry. Applications should make sure that keyboard geometry has no internal conflicts and is consistent with the other components of the keyboard definition, but XKB does not check for or guarantee it.

Querying and Changing Per-Client Flags

XkbPerClientFlags

deviceSpec: KB_DEVICESPEC

change: KB_PCFMASK

value: KB_PCFMASK

ctrlsToChange: KB_BOOLCTRLMASK

autoCtrls: KB_BOOLCTRLMASK

autoCtrlValues: KB_BOOLCTRLMASK

deviceID: CARD8 supported: KB_PCFMASK value: KB_PCFMASK autoCtrls: KB_BOOLCTRLMASK autoCtrlValues: KB_BOOLCTRLMASK where: KB_PCFMASK:

Errors: *Keyboard* , *Value* , *Match* , *Alloc*

Changes the client specific flags for the keyboard specified by *deviceSpec* . Reports a *Keyboard* error if *deviceSpec* does not specify a valid keyboard device.

Any flags specified in *change* are set to the corresponding values in *value* , provided that the server supports the requested control. Legal per-client-flags are:

Flag...	**Described in...**
XkbPCF_DetectableAutorepeat	Detectable Autorepeat
XkbPCF_GrabsUseXKBStateMask	Setting a Passive Grab for an XKB State
XkbPCF_AutoResetControlsMask	Automatic Reset of Boolean Controls
XkbPCF_LookupStateWhenGrabbed	Effects of XKB on Core Protocol Events
XkbPCF_SendEventUsesXKBState	Sending Events to Clients

If *PCF_AutoResetControls* is set in both *change* and *value* , the client's mask of controls to be changed is updated from *ctrlsToChange* , *autoCtrls* , and *autoCtrlValues* . Any controls specified in *ctrlsToChange* are modified in the auto-reset controls mask for the client; the corresponding bits from the *autoCtrls* field are copied into the auto-reset controls mask and the corresponding bits from *autoCtrlValues* are copied into the auto-reset controls state values. If any controls are specified in *autoCtrlValues* but not in *autoCtrls* , a *Match* error results. If any controls are specified in *autoCtrls* but not in *ctrlsToChange* , a *Match* error results.

If *PCF_AutoResetControls* is set in *change* but not in *value* , the client’s mask of controls to be changed is reset to all zeroes (i.e. the client does not change any controls when it exits).

This request reports a *Match* error if a bit is set in any of the value masks but not in the control mask that governs it or a *Value* error if any undefined bits are set in any of the masks.

On successful return, the *deviceID* field reports the X Input extension identifier of the keyboard, or *0* if the server does not support the X Input Extension.

The *supported* return value reports the set of per-client flags that are supported by the server; in this version of XKB, only the *XkbPCF_DetectableAutorepeat* per-client flag is optional; all other per-client flags must be supported.

The *value* return value reports the current settings of all per-client flags for the specified keyboard. The *autoCtrls* return value reports the current set of controls to be reset when the client exits, while the *autoCtrlValues* return value reports the state to which they should be set.

Using the Server’s Database of Keyboard Components

XkbListComponents

deviceSpec: KB_DEVICESPEC

maxNames: CARD16

keymapsSpec: STRING8

keycodesSpec: STRING8

typesSpec: STRING8

compatMapSpec: STRING8

symbolsSpec: STRING8

geometrySpec: STRING8

deviceID: CARD8

extra: CARD16

keymaps,keycodes,types,compatMaps: LISTofKB_COMPONENTNAME

symbols, geometries: LISTofKB_COMPONENTNAME

Where:

KB_COMPONENTNAME { hints: CARD8, name: STRING8 }

Errors: *Keyboard* , *Alloc*

This request returns one or more lists of keyboard components that are available from the X server database of keyboard components for the device specified by *deviceSpec* . The X server is allowed, but not required or expected, to maintain separate databases for each keyboard device. A *Keyboard* error results if *deviceSpec* does not specify a valid keyboard device.

The *maxNames* field specifies the maximum number of component names to be reported, in total, by this request.

The *keymapsSpec* , *keycodesSpec* , *typesSpec* , *compatMapSpec* , *symbolsSpec* and *geometrySpec* request fields specify a pattern to be matched against the names of all components of the corresponding type in the server database of keyboard components.

Each pattern uses the ISO Latin-1 encoding and should contain only parentheses, the wildcard characters "?" and "*" or characters that are permitted in a component class or member name (see Component Names). Illegal characters in a pattern are simply ignored; no error results if a pattern contains illegal characters.

Comparison is case-sensitive and, in a pattern, the "?" wildcard character matches any single character except parentheses while the "*" character matches any number of characters except parentheses. If an implementation accepts characters other than those required by XKB, whether or not those characters match either wildcard is also implementation dependent. An empty pattern does not match any component names.

On successful return, the *deviceID* return value reports the X Input Extension device identifier of the specified device, or *0* if the server does not support the X input extension. The *extra* return value reports the number of matching component names that could not be returned due to the setting of the *maxNames* field in the request.

The *keymaps* , *keycodes* , *types* , *compatMaps* , *symbols* and *geometries* return the hints (see Component Hints) and names of any components from the server database that match the corresponding pattern.

The Server Database of Keyboard Components describes the X server database of keyboard components in more detail.

XkbGetKbdByName

deviceSpec: KB_DEVICESPEC

need, want: KB_GBNDETAILMASK

load: BOOL

keymapsSpec: STRING8

keycodesSpec, typesSpec: STRING8

compatMapSpec, symbolsSpec: STRING8

geometrySpec: STRING8

deviceID: CARD8

minKeyCode, maxKeyCode: KEYCODE

loaded, newKeyboard: BOOL

found, reported: KB_GBNDETAILMASK

map: optional *XkbGetMap* reply

compat: optional *XkbGetCompatMap* reply

indicators: optional *XkbGetIndicatorMap* reply

names: optional *XkbGetNames* reply

geometry: optional *XkbGetGeometry* reply

Errors: *Keyboard* , *Access* , *Alloc*

Assembles and returns a keymap from the current mapping and specified elements from the server database of keymap components for the keyboard specified by *de-*

viceSpec , and optionally replaces the current keyboard mapping with the newly generated description. If *deviceSpec* does not specify a valid keyboard device, a *Keyboard* error results.

The *keymapsSpec* , *keycodesSpec* , *typesSpec* , *compatMapSpec* , *symbolsSpec* and *geometrySpec* component expressions (see Partial Components and Combining Multiple Components) specify the database components to be used to assemble the keyboard description.

The *want* field lists the pieces of the keyboard description that the client wants to have reported for the newly constructed keymap. The *need* field lists all of the pieces that must be reported. If any of the pieces in *need* cannot be loaded from the specified names, no description of the keyboard is returned.

The *want* and *need* fields can include any combinations of these *XkbGetMapByName* (GBN) components:

XkbGetMapByName Keyboard Component…	Database Component…	Components of Keyboard Description
XkbGBN_Types	types	key types
XkbGBN_CompatMap	compat	symbol interpretations, group compatibility map
XkbGBN_ClientSymbols	symbols, types, keycodes	key types, key symbol mappings, modifier mapping
XkbGBN_ServerSymbols	symbols, types, keycodes	key behaviors, key actions, key explicit components, virtual modifiers, virtual modifier mapping
XkbGBN_IndicatorMap	compat	indicator maps, indicator names
XkbGBN_KeyNames	keycodes	key names, key aliases
XkbGBN_Geometry	geometry	keyboard geometry
XkbGBN_OtherNames	all	key types, symbol interpretations, indicator maps, names, geometry

If either field contains a GBN component that depends on some database component for which the request does not supply an expression, XKB automatically substitutes the special pattern "%" which copies the corresponding component from the current keyboard description, as described in Partial Components and Combining Multiple Components.

The *load* flag asks the server to replace the current keyboard description for *deviceSpec* with the newly constructed keyboard description. If *load* is *True* , the request must include component expressions for all of the database components; if any are missing, XKB substitutes "%" as described above.

If all necessary components are both specified and found, the new keyboard description is loaded. If the new keyboard description has a different geometry or keycode range than the previous keyboard description, XKB sends *XkbNewKeyboardNotify* events to all interested clients. See Replacing the Keyboard "On-the-Fly" for more information about the effects of replacing the keyboard description on the fly.

If the range of keycodes changes, clients that have requested *XkbNewKeyboardNotify* events are not sent any other change notification events by this request. Clients that do not request *XkbNewKeyboardNotify* events are sent other XKB change notification events (e.g. *XkbMapNotify* , *XkbNamesNotify*) as necessary to alert them to as many of the keyboard changes as possible.

If no error occurs, the request reply reports the GBN components that were found and sends a description of any of the resulting keyboard that includes and of the components that were requested.

The *deviceID* return value reports the X Input extension device identifier of the keyboard that was used, or *0* if the server does not support the X input extension.

The *minKeyCode* and *maxKeyCode* return values report the legal range of keycodes for the keyboard description that was created. If the resulting keyboard description does not include at least one of the key names, client symbols or server symbols components, *minKeyCode* and *maxKeyCode* are both *0* .

The *loaded* return value reports whether or not the existing keyboard definition was replaced with the newly created one. If *loaded* is *True* , the *newKeyboard* return value reports whether or not the new map changed the geometry or range of keycodes and caused *XkbNewKeyboardNotify* events for clients that have requested them.

The *found* return value reports the GBN components that were present in the keymap that was constructed by this request. The *reported* return value lists the subset of those components for which descriptions follow. if any of the components specified in the *need* field of the request were not found, *reported* is empty, otherwise it contains the intersection of the *found* return value with the union of the *need* and *want* request fields.

If any of *GBN_Types* , *GBN_ClientSymbols* or *GBN_ServerSymbols* are set in *reported* , the *map* return value has the same format as the reply to an *XkbGetMap* request and reports the corresponding pieces of the newly constructed keyboard description.

If *GBN_CompatMap* is set in *reported* , the *compat* return value has the same format as the reply to an *XkbGetCompatMap* request and reports the symbol interpretations and group compatibility map for the newly constructed keyboard description.

If *GBN_IndicatorMap* is set in *reported* , the *indicators* return value has the same format as the reply to an *XkbGetIndicatorMap* request and reports the physical indicators and indicator maps for the newly constructed keyboard description.

If *GBN_KeyNames* or *GBN_OtherNames* are set in *reported* , the *names* return value has the same format as the reply to an *XkbGetNames* reply and reports the corresponding set of symbolic names for the newly constructed keyboard description.

If *GBN_Geometry* is set in *reported* , the *geometry* return value has the same format as the reply to an *XkbGetGeometryMap* request and reports the keyboard geometry for the newly constructed keyboard description.

Querying and Changing Input Extension Devices

XkbGetDeviceInfo
deviceSpec: KB_DEVICESPEC
wanted: KB_XIDEVFEATUREMASK
ledClass: KB_LEDCLASSSPEC
ledID: KB_IDSPEC
allButtons: BOOL
firstButton, nButtons: CARD8
deviceID: CARD8
present: KB_XIDEVFEATUREMASK
supported: KB_XIFEATUREMASK
unsupported: KB_XIFEATUREMASK
firstBtnWanted: CARD8
nBtnsWanted: CARD8
firstBtnRtrn: CARD8
nBtnsRtrn: CARD8
totalBtns: CARD8
hasOwnState: BOOL
dfltKbdFB, dfltLedFB: KB_IDSPEC
devType: ATOM
name: STRING
btnActions: LISTofKB_ACTION
leds: LISTofKB_DEVICELEDINFO
Errors: *Device* , *Match* , *Access* , *Alloc*

Reports a subset of the XKB-supplied information about the input device specified by *deviceSpec* . Unlike most XKB requests, the device specified for *XkbGetDeviceInfo* need not be a keyboard device. Nonetheless, a *Keyboard* error results if *deviceSpec* does not specify a valid core or input extension device.

The *wanted* field specifies the types of information to be returned, and controls the interpretation of the other request fields.

If the server does not support assignment of XKB actions to extension device buttons, the *allButtons* , *firstButton* and *nButtons* fields are ignored.

Otherwise, if the *XkbXI_ButtonActions* flag is set in *wanted* , the *allButtons* , *firstButton* and *nButtons* fields specify the device buttons for which actions should be returned. Setting *allButtons* to *True* requests actions for all device buttons; if *allButtons* is *False* , *firstButton* and *nButtons* specify a range of buttons for which actions are requested. If the device has no buttons or if *firstButton* and *nButtons* specify illegal buttons, a *Match* error results. If *allButtons* is *True* , *firstButton* and *nButtons* are ignored.

If the server does not support XKB access to any aspect of the indicators on extension devices, or if the *wanted* field does not include any of the indicator flags, the *ledClass* and *ledID* fields are ignored. Otherwise, *ledClass* and *ledID* specify one or more feedback(s) for which indicator information is requested. If *ledClass* or *ledID* have illegal values, a *Value* error results. If they have legal values but do not specify a keyboard or indicator class feedback for the device in question, a *Match* error results.

The *ledClass* field can specify either *KbdFeedbackClass* , *LedFeedbackClass* , *XkbDfltXIClass* , or *XkbAllXIClasses* . If at least one keyboard feedback is defined for the specified device, *XkbDfltXIClass* is equivalent to *KbdFeedbackClass* , otherwise it is equivalent to *LedFeedbackClass* . If *XkbAllXIClasses* is specified, this request returns information about both indicator and keyboard class feedbacks which match the requested identifier, as described below.

The *ledID* field can specify any valid input extension feedback identifier, *XkbDfltXIId* , or *XkbAllXIIds* . The default keyboard feedback is the one that is affected by core protocol requests; the default led feedback is implementation-specific. If *XkbAllXIIds* is specified, this request returns indicator information about all feedbacks of the class(es) specified by *ledClass* .

If no error results, the *deviceID* return value reports the input extension device identifier of the device for which values are being returned. The *supported* return value reports the set of optional XKB extension device features that are supported by this implementation (see Interactions Between XKB and the X Input Extension) for the specified device, and the unsupported return value reports any *unsupported* features.

If *hasOwnState* is *True* , the device is also a keyboard, and any indicator maps bound to the device use the current state and control settings for this device to control automatic changes. If *hasOwnState* is *False* , the state and control settings of the core keyboard device control automatic indicator changes.

The *name* field reports the X Input Extension name for the device. The *devType* field reports the X Input Extension device type. Both fields are provided merely for convenience and are not interpreted by XKB.

The *present* return value reports the kinds of device information being returned, and controls the interpretation of the remaining fields. The *present* field consists of the *wanted* field from the original request minus the flags for any unsupported features.

If *XkbXI_ButtonActions* is set in *present* , the *totalBtns* return value reports the total number of buttons present on the device, *firstBtnWanted* and *nBtnsWanted* specify the range of buttons for which actions were requested, and the *firstBtnRtrn* and *nBtnsRtrn* values specify the range of buttons for which actions are reported. The *actionsRtrn* list has *nButtonsRtrn* entries which contain the actions bound to the specified buttons on the device. Any buttons for which actions were requested but not returned have the action *NoAction()* .

If any indicator information is reported, the leds list contains one element for each requested feedback. For example, if *ledClass* is *XkbAllXIClasses* and *ledID* is *XkbAllXIIds* , *leds* describes all of the indicators on the device and has one element for each keyboard or led class feedback defined for the device. If any information at all is reported about a feedback, the set of physical indicators is also reported in the *physIndicators* field of the corresponding element of *leds* .

If the server supports assignment of indicator maps to extension device indicators, and if the *XkbXI_IndicatorMaps* flag is set in *wanted* , each member of *leds* reports any indicators on the corresponding feedback to which names have been assigned. Any indicators for which no map is reported have the default map, which allows explicit changes and does not request any automatic changes.

If the server supports assignment of indicator names to extension device indicators, and the *XkbXI_IndicatorNames* flag is set in *wanted* , each member of *leds* reports any indicators on the corresponding feedback to which names have been assigned. Any indicators for which no name is reported have the name *None* .

If the server supports XKB access to the state of extension device indicators, and the *XkbXI_IndicatorState* flag is set in wanted, each member of leds reports the state of the indicators on the corresponding feedback.

If any unsupported features are requested, and the requesting client has selected for them, the server sends the client an *XkbExtensionDeviceNotify* event which indicates that an unsupported feature was requested. This event is only generated if the client which issued the unsupported request has selected for it and, if generated, is not sent to any other clients.

XkbSetDeviceInfo

deviceSpec: KB_DEVICESPEC

change: KB_XIDEVFEATUREMASK

firstBtn, nBtns: CARD8

btnActions:LISTofKB_ACTION

leds: LISTofKB_DEVICELEDINFO

Errors: *Device* , *Match* , *Access* , *Alloc*

Changes a subset of the XKB-supplied information about the input device specified by *deviceSpec* . Unlike most XKB requests, the device specified for *XkbGetDeviceInfo* need not be a keyboard device. Nonetheless, a *Keyboard* error results if *deviceSpec* does not specify a valid core or input extension device

The *change* field specifies the features for which new values are supplied, and controls the interpretation of the other request fields.

If the server does not support assignment of XKB actions to extension device buttons, the *firstButton* and *nButtons* fields are ignored.

Otherwise, if the *XkbXI_ButtonActions* flag is set in *change* , the *firstBtn* and *nBtns* fields specify a range of buttons for which actions are specified in this request. If the device has no buttons or if *firstBtn* and *nBtns* specify illegal buttons, a *Match* error results.

Each element of the *leds* list describes the changes for a single keyboard or led feedback. If the *ledClass* field of any element of *leds* contains any value other than *KbdFeedbackClass* , *LedFeedbackClass* or *XkbDfltXIClass* , a *Value* error results. If the *ledId* field of any element of leds contains any value other than a valid input extension feedback identifier or *XkbDfltXIId* , a *Value* error results. If both fields are valid, but the device has no matching feedback, a *Match* error results.

The fields of each element of *leds* are interpreted as follows:

- If *XkbXI_IndicatorMaps* is set in *change* and the server supports XKB assignment of indicator maps to the corresponding feedback, the maps for all indicators on the corresponding feedback are taken from *leds* . If the server does not support this feature, any maps specified in *leds* are ignored.
- If *XkbXI_IndicatorNames* is set in *change* , and the server supports XKB assignment of names to indicators for the corresponding feedback, the names for all indicators on the corresponding feedback are taken from *leds* . If the server does not support this feature, any names specified in *leds* are ignored. Regardless of whether they are used, any names be a valid Atom or *None* , or an *Atom* error results.
- If *XkbXI_IndicatorState* is set in change, and the server supports XKB changes to extension device indicator state, the server attempts to change the indicators on the corresponding feedback as specified by *leds* . Any indicator maps bound to the feedback are applied, so state changes might be blocked or have side-effects.

If any unsupported features are requested, and the requesting client has selected for them, the server sends the client an *XkbExtensionDeviceNotify* event which indicates that an unsupported feature was requested. This event is only generated if the client which issued the unsupported request has selected for it and, if generated, is not sent to any other clients.

Debugging the X Keyboard Extension

XkbSetDebuggingFlags

affectFlags, flags: CARD32

affectCtrls, ctrls: CARD32

message: STRING

currentFlags, supportedFlags: CARD32

currentCtrls, supportedCtrls: CARD32

This request sets up various internal XKB debugging flags and controls. It is intended for developer use and may be disabled in production servers. If disabled, *XkbSetDebuggingFlags* has no effect but returns *Success* .

The *affectFlags* field specifies the debugging flags to be changed, the *flags* field specifies new values for the changed flags. The interpretation of the debugging flags is implementation-specific, but flags are intended to control debugging output and should not otherwise affect the operation of the server.

The *affectCtrls* field specifies the debugging controls to be changed, the *ctrls* field specifies new values for the changed controls. The interpretation of the debugging controls is implementation-specific, but debugging controls are allowed to affect the behavior of the server.

The *message* field provides a message that the X server can print in any logging or debugging files before changing the flags. The server must accept this field but it is not required to actually display it anywhere.

The X Test Suite makes some assumptions about the implementation of locking modifier keys that do not apply when XKB is present. The *XkbDF_DisableLocks* debugging control provides a simple workaround to these test suite problems by simply disabling all locking keys. If *XkbDF_DisableLocks* is enabled, the *SA_LockMods* and *SA_LockGroup* actions behave like *SA_SetMods* and *SA_LockMods* , respectively. If it is disabled, *SA_LockMods* and *SA_LockGroup* actions behave normally.

Implementations are free to ignore the *XkbDF_DisableLocks* debugging control or to define others.

The *currentFlags* return value reports the current setting for the debugging flags, if applicable. The *currentCtrls* return value reports the setting for the debugging controls, if applicable. The *supportedFlags* and *supportedCtrls* fields report the flags and controls that are recognized by the implementation. Attempts to change unsupported fields or controls are silently ignored.

If the *XkbSetDebuggingFlags* request contains more data than expected, the server ignores the extra data, but no error results. If the request has less data than expected, a *Length* error results.

If the *XkbSetDebuggingFlags* reply contains more data than expected, the client just ignores any uninterpreted data without reporting an error. If the reply has less data than expected, a *Length* error results.

Events

All XKB events report the time at which they occurred in a field named *time* and the device on which they occurred in a field named *deviceID* . XKB uses a single X event code for all events and uses a common field to distinguish XKB event type.

Tracking Keyboard Replacement

XkbNewKeyboardNotify
time: TIMESTAMP
deviceID: CARD8
changed: KB_NKNDETAILMASK
minKeyCode, maxKeyCode: KEYCODE
oldDeviceID: CARD8
oldMinKeyCode, oldMaxKeyCode: KEYCODE
requestMajor, requestMinor: CARD8

An *XkbNewKeyboardNotify* event reports that a new core keyboard has been installed. New keyboard notify events can be generated:

- When the X server detects that the keyboard was changed.
- When a client installs a new extension device as the core keyboard using the X Input Extension *ChangeKeyboardDevice* request.
- When a client issues an *XkbGetMapByName* request which changes the keycodes range or geometry.

The *changed* field of the event reports the aspects of the keyboard that have changed, and can contain any combination of the event details for this event:

Bit in Changed	Meaning
NKN_Keycodes	The new keyboard has a different minimum or maximum keycode.
NKN_Geometry	The new keyboard has a different keyboard geometry.
NKN_DeviceID	The new keyboard has a new X Input Extension device identifier

The server sends an *XkbNewKeyboardNotify* event to a client only if at least one of the bits that is set in the *changed* field of the event is also set in the appropriate event details mask for the client.

The *minKeyCode* and *maxKeyCode* fields report the minimum and maximum keycodes that can be returned by the new keyboard. The *oldMinKeyCode* and *oldMaxKeyCode* fields report the minimum and maximum values that could be returned before the change. This event always reports all four values, but the old and new values are the same unless *NKN_Keycodes* is set in *changed* .

Once a client receives a new keyboard notify event which reports a new keycode range, the X server reports events from all keys in the new range to that client. Clients that do not request or receive new keyboard notify events receive events only from keys that fall in the last range for legal keys reported to that client. See Replacing the Keyboard "On-the-Fly" for a more detailed explanation.

If *NKN_Keycodes* is set in *changed* , the *XkbNewKeyboardNotify* event subsumes all other change notification events (e.g. *XkbMapNotify* , *XkbNamesNotify*) that would otherwise result from the keyboard change. Clients who receive an *XkbNewKeyboardNotify* event should assume that all other aspects of the keyboard mapping have changed and regenerate the entire local copy of the keyboard description.

The *deviceID* field reports the X Input Extension device identifier of the new keyboard device; *oldDeviceID* reports the device identifier before the change. This event always includes both values, but they are the same unless *NKN_DeviceID* is set in *changed* . If the server does not support the X Input Extension, both fields have the value *0* .

The *requestMajor* and *requestMinor* fields report the major and minor opcode of the request that caused the keyboard change. If the keyboard change was not caused by some client request, both fields have the value *0* .

Tracking Keyboard Mapping Changes

XkbMapNotify
time: TIMESTAMP
deviceID: CARD8
ptrBtnActions: CARD8
changed: KB_MAPPARTMASK
minKeyCode, maxKeyCode: KEYCODE
firstType, nTypes: CARD8
firstKeySym, firstKeyAction: KEYCODE
nKeySyms, nKeyActions: CARD8
firstKeyBehavior, firstKeyExplicit: KEYCODE
nKeyBehaviors, nKeyExplicit: CARD8
virtualMods: KB_VMODMASK
firstModMapKey, firstVModMapKey: KEYCODE
nModMapKeys, nVModMapKeys: CARD8

An *XkbMapNotify* event reports that some aspect of XKB map for a keyboard has changed. Map notify events can be generated whenever some aspect of the keyboard map is changed by an XKB or core protocol request.

The *deviceID* field reports the keyboard for which some map component has changed and the *changed* field reports the components with new values, and can contain any of the values that are legal for the *full* and *partial* fields of the *XkbGetMap* request. The server sends an *XkbMapNotify* event to a client only if at least one of the bits that is set in the *changed* field of the event is also set in the appropriate event details mask for the client.

The *minKeyCode* and *maxKeyCode* fields report the range of keycodes that are legal on the keyboard for which the change is being reported.

If *XkbKeyTypesMask* is set in *changed* , the *firstType* and *nTypes* fields report a range of key types that includes all changed types. Otherwise, both fields are *0* .

If *XkbKeySymsMask* is set in *changed* , the *firstKeySym* and *nKeySyms* fields report a range of keycodes that includes all keys with new symbols. Otherwise, both fields are *0* .

If *XkbKeyActionsMask* is set in *changed* , the *firstKeyAction* and *nKeyActions* fields report a range of keycodes that includes all keys with new actions. Otherwise, both fields are *0* .

If *XkbKeyBehaviorsMask* is set in *changed* , the *firstKeyBehavior* and *nKeyBehaviors* fields report a range of keycodes that includes all keys with new key behavior. Otherwise, both fields are *0* .

If *XkbVirtualModsMask* is set in *changed* , *virtualMods* contains all virtual modifiers to which a new set of real modifiers is bound. Otherwise, *virtualMods* is *0* .

If *XkbExplicitComponentsMask* is set in *changed* , the *firstKeyExplicit* and *nKeyExplicit* fields report a range of keycodes that includes all keys with changed explicit components. Otherwise, both fields are *0* .

If *XkbModifierMapMask* is set in *changed* , the *firstModMapKey* and *nModMapKeys* fields report a range of keycodes that includes all keys with changed modifier bindings. Otherwise, both fields are *0* .

If *XkbVirtualModMapMask* is set in *changed* , the *firstVModMapKey* and *nVModMapKeys* fields report a range of keycodes that includes all keys with changed virtual modifier mappings. Otherwise, both fields are *0* .

Tracking Keyboard State Changes

XkbStateNotify

time: TIMESTAMP

deviceID: CARD8

mods, baseMods, latchedMods, lockedMods: KEYMASK

group, lockedGroup: CARD8

baseGroup, latchedGroup: INT16

compatState: KEYMASK

grabMods, compatGrabMods: KEYMASK

lookupMods, compatLookupMods: KEYMASK

ptrBtnState: BUTMASK

changed: KB_STATEPARTMASK

keycode: KEYCODE

eventType: CARD8

requestMajor, requestMinor: CARD8

An *XkbStateNotify* event reports that some component of the XKB state (see Keyboard State) has changed. State notify events are usually caused by key or pointer activity, but they can also result from explicit state changes requested by the *XkbLatchLockState* request or by other extensions.

The *deviceID* field reports the keyboard on which some state component changed. The *changed* field reports the XKB state components (see Keyboard State) that have changed and contain any combination of:

Bit in changed	Event field	Changed component
ModifierState	*mods*	The effective modifiers
ModifierBase	*baseMods*	The base modifiers
ModifierLatch	*latchedMods*	The latched modifiers
ModifierLock	*lockedMods*	The locked modifiers
GroupState	*group*	The effective keyboard group
GroupBase	*baseGroup*	The base keyboard group
GroupLatch	*latchedGroup*	The latched keyboard group
GroupLock	*lockedGroup*	The locked keyboard group
PointerButtons	*ptrBtnState*	The state of the core pointer buttons
GrabMods	*grabMods*	The XKB state used to compute grabs
LookupMods	*lookupMods*	The XKB state used to look up symbols
CompatState	*compatState*	Default state for non-XKB clients
CompatGrabMods	*compatGrabMods*	The core state used to compute grabs
CompatLookupMods	*compatLookupMods*	The core state used to look up symbols

The server sends an *XkbStateNotify* event to a client only if at least one of the bits that is set in the *changed* field of the event is also set in the appropriate event details mask for the client.

A state notify event reports current values for all state components, even those with unchanged values.

The *keycode* field reports the key or button which caused the change in state while the *eventType* field reports the exact type of event (e.g. *KeyPress*). If the change in state was not caused by key or button activity, both fields have the value *0* .

The *requestMajor* and *requestMinor* fields report the major and minor opcodes of the request that caused the change in state and have the value *0* if it was resulted from key or button activity.

Tracking Keyboard Control Changes

XkbControlsNotify

time: TIMESTAMP

deviceID: CARD8

numGroups: CARD8

changedControls: KB_CONTROLMASK

enabledControls,enabledControlChanges: KB_BOOLCTRLMASK

keycode: KEYCODE

eventType: CARD8

requestMajor: CARD8

requestMinor: CARD8

An *XkbControlsNotify* event reports a change in one or more of the global keyboard controls (see Global Keyboard Controls) or in the internal modifiers or ignore locks masks (see Server Internal Modifiers and Ignore Locks Behavior). Controls notify events are usually caused by and *XkbSetControls* request, but they can also be caused by keyboard activity or certain core protocol and input extension requests.

The *deviceID* field reports the keyboard for which some control has changed, and the *changed* field reports the controls that have new values.

The *changed* field can contain any of the values that are permitted for the *changeControls* field of the *XkbSetControls* request. The server sends an *XkbControlsNotify* event to a client only if at least one of the bits that is set in the *changed* field of the event is also set in the appropriate event details mask for the client.

The *numGroups* field reports the total number of groups defined for the keyboard, whether or not the number of groups has changed.

The *enabledControls* field reports the current status of all of the boolean controls, whether or not any boolean controls changed state. If *EnabledControls* is set in *changed* , the *enabledControlChanges* field reports the boolean controls that were enabled or disabled; if a control is specified in *enabledControlChanges* , the value that is reported for that control in *enabledControls* represents a change in state.

The *keycode* field reports the key or button which caused the change in state while the *eventType* field reports the exact type of event (e.g. *KeyPress*). If the change in state was not caused by key or button activity, both fields have the value *0* .

The *requestMajor* and *requestMinor* fields report the major and minor opcodes of the request that caused the change in state and have the value *0* if it was resulted from key or button activity.

Tracking Keyboard Indicator State Changes

XkbIndicatorStateNotify

time: TIMESTAMP

deviceID: CARD8

stateChanged, state: KB_INDICATORMASK

An *XkbIndicatorStateNotify* event indicates that one or more of the indicators on a keyboard have changed state. Indicator state notify events can be caused by:

- Automatic update to reflect changes in keyboard state (keyboard activity, *XkbLatchLockState* requests).
- Automatic update to reflect changes in keyboard controls (*XkbSetControls* , keyboard activity, certain core protocol and input extension requests).
- Explicit attempts to change indicator state (core protocol and input extension requests, *XkbSetNamedIndicator* requests).
- Changes to indicator maps (*XkbSetIndicatorMap* and *XkbSetNamedIndicator* requests).

The *deviceID* field reports the keyboard for which some indicator has changed, and the *state* field reports the new state for all indicators on the specified keyboard. The *stateChanged* field specifies which of the values in *state* represent a new state for the corresponding indicator. The server sends an *XkbIndicatorStateNotify* event to a client only if at least one of the bits that is set in the *stateChanged* field of the event is also set in the appropriate event details mask for the client.

Tracking Keyboard Indicator Map Changes

XkbIndicatorMapNotify

time: TIMESTAMP

deviceID: CARD8

state: KB_INDICATORMASK

mapChanged: KB_INDICATORMASK

An *XkbIndicatorMapNotify* event indicates that the maps for one or more keyboard indicators have been changed. Indicator map notify events can be caused by *XkbSetIndicatorMap* and *XkbSetNamedIndicator* requests.

The *deviceID* field reports the keyboard for which some indicator map has changed, and the *mapChanged* field reports the indicators with changed maps. The server sends an *XkbIndicatorMapNotify* event to a client only if at least one of the bits that is set in the *mapChanged* field of the event is also set in the appropriate event details mask for the client.

The *state* field reports the current state of all indicators on the specified keyboard.

Tracking Keyboard Name Changes

XkbNamesNotify

time: TIMESTAMP

deviceID: CARD8

changed: KB_NAMEDETAILMASK

firstType, nTypes: CARD8

firstLevelName, nLevelNames: CARD8

firstKey: KEYCODE

nKeys, nKeyAliases, nRadioGroups: CARD8

changedGroupNames: KB_GROUPMASK

changedVirtualMods: KB_VMODMASK

changedIndicators: KB_INDICATORMASK

An *XkbNamesNotify* event reports a change to one or more of the symbolic names associated with a keyboard. Symbolic names can change when:

- Some client explicitly changes them using *XkbSetNames* .
- The list of key types or radio groups is resized
- The group width of some key type is changed

The *deviceID* field reports the keyboard on which names were changed. The *changed* mask lists the components for which some names have changed and can have any combination of the values permitted for the *which* field of the *XkbGetNames* request. The server sends an *XkbNamesNotify* event to a client only if at least one of the bits that is set in the *changed* field of the event is also set in the appropriate event details mask for the client.

If *KeyTypeNames* is set in *changed* , the *firstType* and *nTypes* fields report a range of types that includes all types with changed names. Otherwise, both fields are *0* .

If *KTLevelNames* is set in *changed* , the *firstLevelName* and *nLevelNames* fields report a range of types that includes all types with changed level names. Otherwise, both fields are *0* .

If *IndicatorNames* is set in *changed* , the *changedIndicators* field reports the indicators with changed names. Otherwise, *changedIndicators* is *0* .

If *VirtualModNames* is set in *changed* , the *changedVirtualMods* field reports the virtual modifiers with changed names. Otherwise, *changedVirtualMods* is *0* .

If *GroupNames* is set in *changed* , the *changedGroupNames* field reports the groups with changed names. Otherwise, *changedGroupNames* is *0* .

If *KeyNames* is set in *changed* , the *firstKey* and *nKeys* fields report a range of keycodes that includes all keys with changed names. Otherwise, both fields are *0* .

The *nKeyAliases* field reports the total number of key aliases associated with the keyboard, regardless of whether *KeyAliases* is set in *changed* .

The *nRadioGroups* field reports the total number of radio group names associated with the keyboard, regardless of whether *RGNames* is set in *changed* .

Tracking Compatibility Map Changes

XkbCompatMapNotify

time: TIMESTAMP

deviceID: CARD8

changedGroups: KB_GROUPMASK

firstSI, nSI: CARD16

nTotalSI: CARD16

An *XkbCompatMapNotify* event indicates that some component of the compatibility map for a keyboard has been changed. Compatibility map notify events can be caused by *XkbSetCompatMap* and *XkbGetMapByName* requests.

The *deviceID* field reports the keyboard for which the compatibility map has changed; if the server does not support the X input extension, *deviceID* is *0* .

The *changedGroups* field reports the keyboard groups, if any, with a changed entry in the group compatibility map. The *firstSI* and *nSI* fields specify a range of symbol interpretations in the symbol compatibility map that includes all changed symbol interpretations; if the symbol compatibility map is unchanged, both fields are *0* . The *nTotalSI* field always reports the total number of symbol interpretations present in the symbol compatibility map, regardless of whether any symbol interpretations have been changed.

The server sends an *XkbCompatMapNotify* event to a client only if at least one of the following conditions is met:

- The *nSI* field of the event is non-zero, and the *XkbSymInterpMask* bit is set in the appropriate event details mask for the client.
- The *changedGroups* field of the event contains at least one group, and the *XkbGroupCompatMask* bit is set in the appropriate event details mask for the client.

Tracking Application Bell Requests

XkbBellNotify
time: TIMESTAMP
deviceID: CARD8
bellClass: { KbdFeedbackClass, BellFeedbackClass }
bellID: CARD8
percent: CARD8
pitch: CARD16
duration: CARD16
eventOnly: BOOL
name: ATOM
window: WINDOW

An *XkbBellNotify* event indicates that some client has requested a keyboard bell. Bell notify events are usually caused by *Bell* , *DeviceBell* , or *XkbBell* requests, but they can also be generated by the server (e.g. if the *AccessXFeedback* control is active).

The server sends an *XkbBellNotify* event to a client if the appropriate event details field for the client has the value *True* .

The *deviceID* field specifies the device for which a bell was requested, while 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, all three fields have the value 0.

The *percent* , *pitch* and *duration* fields report the volume, tone and duration requested for the bell as specified by the *XkbBell* request. Bell notify events caused by core protocol or input extension requests use the pitch and duration specified in the corresponding bell or keyboard feedback control.

If the bell was caused by an *XkbBell* request or by the X server, *name* reports an optional symbolic name for the bell and the *window* field optionally reports the window for which the bell was generated. Otherwise, both fields have the value *None* .

If the *eventOnly* field is *True* , the server did not generate a sound in response to the request, otherwise the server issues the beep before sending the event. The eventOnly field can be *True* if the *AudibleBell* control is disabled or if a client explicitly requests *eventOnly* when it issues an *XkbBell* request.

Tracking Messages Generated by Key Actions

XkbActionMessage
time: TIMESTAMP
deviceID: CARD8
keycode: KEYCODE
press: BOOL
mods: KEYMASK
group: KB_GROUP
keyEventFollows: BOOL
message: LISTofCARD8

An *XkbActionMessage* event is generated when the user operates a key to which an *SA_ActionMessage* message is bound under the appropriate state and group. The server sends an *XkbActionMessage* event to a client if the appropriate event details field for the client has the value *True* .

The *deviceID* field specifies the keyboard device that contains the key which activated the event. The *keycode* field specifies the key whose operation caused the message and press is *True* if the message was caused by the user pressing the key. The *mods* and *group* fields report the effective keyboard modifiers and group in effect at the time the key was pressed or released.

If *keyEventFollows* is *True* , the server will also send a key press or release event, as appropriate, for the key that generated the message. If it is *False* , the key causes only a message. Note that the key event is delivered normally with respect to passive grabs, keyboard focus, and cursor position, so that *keyEventFollows* does not guarantee that any particular client which receives the *XkbActionMessage* notify event will also receive a key press or release event.

The *message* field is *NULL* -terminated string of up to *ActionMessageLength* (*6*) bytes, which reports the contents of the *message* field in the action that caused the message notify event.

Tracking Changes to AccessX State and Keys

XkbAccessXNotify
time: TIMESTAMP
deviceID: CARD8
detail: KB_AXNDETAILMASK
keycode: KEYCODE
slowKeysDelay: CARD16
debounceDelay: CARD16

An *XkbAccessXNotify* event reports on some kinds of keyboard activity when any of the *SlowKeys* , *BounceKeys* or *AccessXKeys* controls are active. Compatibility map notify events can only be caused by keyboard activity.

The *deviceID* and *keycode* fields specify the keyboard and key for which the event occurred. The *detail* field describes the event that occurred and has one of the following values:

Detail	Control	Meaning
AXN_SKPress	*SlowKeys*	Key pressed
AXN_SKAccept	*SlowKeys*	*K* ey held until it was accepted.
AXN_SKReject	*SlowKeys*	Key released before it was accepted.
AXN_SKRelease	*SlowKeys*	Key released after it was accepted.
AXN_BKAccept	*BounceKeys*	Key pressed while it was active.
AXN_BKReject	*BounceKeys*	Key pressed while it was still disabled.
AXN_AXKWarning	*AccessXKeys*	Shift key held down for four seconds

Each subclass of the AccessX notify event is generated only when the control specified in the table above is enabled. The server sends an *XkbAccessXNotify* event to a client only if the bit which corresponds to the value of the *detail* field for the event is set in the appropriate event details mask for the client.

Regardless of the value of *detail* , the *slowKeysDelay* and *debounceDelay* fields always reports the current slow keys acceptance delay (see The SlowKeys Control) and debounce delay (see The BounceKeys Control) for the specified keyboard.

Tracking Changes To Extension Devices

XkbExtensionDeviceNotify

time: TIMESTAMP

deviceID: CARD16

ledClass: { KbdFeedbackClass, LedFeedbackClass }

ledID: CARD16

reason: KB_XIDETAILMASK

supported: KB_XIFEATUREMASK

unsupported: KB_XIFEATUREMASK

ledsDefined: KB_INDICATORMASK

ledState: KB_INDICATORMASK

firstButton, nButtons: CARD8

An *XkbExtensionDeviceNotify* event reports:

- A change to some part of the XKB information for an extension device.
- An attempt to use an XKB extension device feature that is not supported for the specified device by the current implementation.

The *deviceID* field specifies the X Input Extension device identifier of some device on which an XKB feature was requested, or *XkbUseCorePtr* if the request affected the core pointer device. The *reason* field explains why the event was generated in response to the request, and can contain any combination of *XkbXI_UnsupportedFeature* and the values permitted for the change field of the *XkbSetDeviceInfo* request.

If *XkbXI_ButtonActions* is set in *reason* , this event reports a successful change to the XKB actions bound to one or more buttons on the core pointer or an extension device. The *firstButton* and *nButtons* fields report a range of device buttons that include all of the buttons for which actions were changed.

If any combination of *XkbXI_IndicatorNames* , *XkbXI_IndicatorMaps* , or *XkbXI_IndicatorState* is set in either *reason* or *unsupported* , the *ledClass* and *ledID* fields specify the X Input Extension feedback class and identifier of the feedback for which the change is reported. If this event reports any changes to an indicator feedback, the *ledsDefined* field reports all indicators on that feedback for which either a name or a indicator map are defined, and *ledState* reports the current state of all of the indicators on the specified feedback.

If *XkbXI_IndicatorNames* is set in *reason* , this event reports a successful change to the symbolic names bound to one or more extension device indicators by XKB. If *XkbXI_IndicatorMaps* is set in *reason* , this event reports a successful change to the indicator maps bound to one or more extension device indicators by XKB. If *XkbXI_IndicatorState* is set in reason, this event reports that one or more indicators in the specified device and feedback have changed state.

If *XkbXI_UnsupportedFeature* is set in reason, this event reports an unsuccessful attempt to use some XKB extension device feature that is not supported by the XKB implementation in the server for the specified device. The *unsupported* mask reports the requested features that are not available on the specified device. See Interactions Between XKB and the X Input Extension for more information about possible XKB interactions with the X Input Extension.

The server sends an *XkbExtensionDeviceNotify* event to a client only if at least one of the bits that is set in the *reason* field of the event is also set in the appropriate event details mask for the client.

Events that report a successful change to some extension device feature are reported to all clients that have expressed interest in the event; events that report an attempt to use an unsupported feature are reported only to the client which issued the request. Events which report a partial success are reported to all interested clients, but only the client that issued the request is informed of the attempt to use unsupported features.

Appendix A. Default Symbol Transformations

Interpreting the Control Modifier

If the *Control* modifier is not consumed by the symbol lookup process, routines that determine the symbol and string that correspond to an event should convert the symbol to a string as defined in the table below. Only the string to be returned is affected by the *Control* modifier; the symbol is not changed.

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

Interpreting the Lock Modifier

If the *Lock* modifier is not consumed by the symbol lookup process, routines that determine the symbol and string that correspond to an event should capitalize the result. Unlike the transformation for *Control* , the capitalization transformation changes both the symbol and the string returned by the event.

Locale-Sensitive Capitalization

If *Lock* is set in an event and not consumed, applications should capitalize the string and symbols that result from an event according to the capitalization rules in effect for the system on which the application is running, taking the current state of the user environment (e.g. locale) into account.

Locale-Insensitive Capitalization

XKB recommends but does not require locale-sensitive capitalization. In cases where the locale is unknown or where locale-sensitive capitalization is prohibitively expensive, applications can capitalize according to the rules defined in this extension.

The following tables list all of the keysyms for which XKB defines capitalization behavior. Any keysyms not explicitly listed in these tables are not capitalized by XKB when locale-insensitive capitalization is in effect and are not automatically assigned the *ALPHABETIC* type as described in the Alphabetic Key Type.

Capitalization Rules for Latin-1 Keysyms

This table lists the Latin-11 keysyms for which XKB defines upper and lower case:

Lower Case	**Upper Case**	**Lower Case**	**Upper Case**	**Lower Case**	**Upper Case**	**Lower Case**	**Upper Case**
a	A	o	O	acircumflex	Acircumflex	eth	ETH
b	B	p	P	adiaeresis	Adiaeresis	ntilde	Ntilde
c	C	q	Q	atilde	Atilde	ograve	Ograve
d	D	r	R	aring	Aring	oacute	Oacute
e	E	s	S	ae	AE	ocircumflex	Ocircumflex
f	F	t	T	ccedilla	Ccedilla	otilde	Otilde
g	G	u	U	egrave	Egrave	odiaeresis	Odiaeresis
h	H	v	V	eacute	Eacute	oslash	Ooblique
i	I	w	W	ecircumflex	Ecircumflex	ugrave	Ugrave
j	J	x	X	ediaeresis	Ediaeresis	uacute	Uacute
k	K	y	Y	igrave	Igrave	ucircumflex	Ucircumflex
l	L	z	Z	iacute	Iacute	udiaeresis	Udiaeresis
m	M	agrave	Agrave	icircumflex	Icircumflex	yacute	Yacute
n	N	aacute	Aacute	idiaeresis	Idiaeresis	thorn	THORN

Capitalization Rules for Latin-2 Keysyms

This table lists the Latin-2 keysyms for which XKB defines upper and lower case:

Lower Case	**Upper Case**	**Lower Case**	**Upper Case**	**Lower Case**	**Upper Case**
aogonek	Aogonek	zabovedot	Zabovedot	dstroke	Dstroke
lstroke	Lstroke	racute	Racute	nacute	Nacute
lcaron	Lcaron	abreve	Abreve	ncaron	Ncaron
sacute	Sacute	lacute	Lacute	odoubleacute	Odoubleacute
scaron	Scaron	cacute	Cacute	rcaron	Rcaron
scedilla	Scedilla	ccaron	Ccaron	uabovering	Uabovering
tcaron	Tcaron	eogonek	Eogonek	udoubleacute	Udoubleacute
zacute	Zacute	ecaron	Ecaron	tcedilla	Tcedilla
zcaron	Zcaron	dcaron	Dcaron		

Capitalization Rules for Latin-3 Keysyms

This table lists the Latin-3 keysyms for which XKB defines upper and lower case:

Lower Case	Upper Case	Lower Case	Upper Case	Lower Case	Upper Case
hstroke	Hstroke	jcircumflex	Jcircumflex	gcircumflex	Gcircumflex
hcircumflex	Hcircumflex	cabovedot	Cabovedot	ubreve	Ubreve
idotless	Iabovedot	ccircumflex	Ccircumflex	scircumflex	Scircumflex
gbreve	Gbreve	gabovedot	Gabovedot		

Capitalization Rules for Latin-4 Keysyms

This table lists the Latin-4 keysyms for which XKB defines upper and lower case:

Lower Case	Upper Case	Lower Case	Upper Case	Lower Case	Upper Case
rcedilla	Rcedilla	eng	ENG	omacron	Omacron
itilde	Itilde	amacron	Amacron	kcedilla	Kcedilla
lcedilla	Lcedilla	iogonek	Iogonek	uogonek	Uogonek
emacron	Emacron	eabovedot	eabovedot	utilde	Utilde
gcedilla	Gcedilla	imacron	Imacron	umacron	Umacron
tslash	Tslash	ncedilla	Ncedilla		

Capitalization Rules for Cyrillic Keysyms

This table lists the Cyrillic keysyms for which XKB defines upper and lower case:

Lower Case	Upper Case	Lower Case	Upper Case
Serbian_dje	Serbian_DJE	Cyrillic_i	Cyrillic_I
Macedonia_gje	Macedonia_GJE	Cyrillic_shorti	Cyrillic_SHORTI
Cyrillic_io	Cyrillic_IO	Cyrillic_ka	Cyrillic_KA
Ukrainian_ie	Ukrainian_IE	Cyrillic_el	Cyrillic_EL
Macedonia_dse	Macedonia_DSE	Cyrillic_em	Cyrillic_EM
Ukrainian_i	Ukrainian_I	Cyrillic_en	Cyrillic_EN
Ukrainian_yi	Ukrainian_YI	Cyrillic_o	Cyrillic_O
Cyrillic_je	Cyrillic_JE	Cyrillic_pe	Cyrillic_PE
Cyrillic_lje	Cyrillic_LJE	Cyrillic_ya	Cyrillic_YA
Cyrillic_nje	Cyrillic_NJE	Cyrillic_er	Cyrillic_ER
Serbian_tshe	Serbian_TSHE	Cyrillic_es	Cyrillic_ES
Macedonia_kje	Macedonia_KJE	Cyrillic_te	Cyrillic_TE
Byelorussian_shortu	Byelorussian_SHORTU	Cyrillic_u	Cyrillic_U
Cyrillic_dzhe	Cyrillic_DZHE	Cyrillic_zhe	Cyrillic_ZHE
Cyrillic_yu	Cyrillic_YU	Cyrillic_ve	Cyrillic_VE
Cyrillic_a	Cyrillic_A	Cyrillic_softsign	Cyrillic_SOFTSIGN
Cyrillic_be	Cyrillic_BE	Cyrillic_yeru	Cyrillic_YERU
Cyrillic_tse	Cyrillic_TSE	Cyrillic_ze	Cyrillic_ZE
Cyrillic_de	Cyrillic_DE	Cyrillic_sha	Cyrillic_SHA
Cyrillic_ie	Cyrillic_IE	Cyrillic_e	Cyrillic_E
Cyrillic_ef	Cyrillic_EF	Cyrillic_shcha	Cyrillic_SHCHA
Cyrillic_ghe	Cyrillic_GHE	Cyrillic_che	Cyrillic_CHE
Cyrillic_ha	Cyrillic_HA	Cyrillic_hardsign	Cyrillic_HARDSIGN

Capitalization Rules for Greek Keysyms

This table lists the Greek keysyms for which XKB defines upper and lower case:

Lower Case	**Upper Case**	**Lower Case**	**Upper Case**
Greek_omegaaccent	Greek_OMEGAACCENT	Greek_iota	Greek_IOTA
Greek_alphaaccent	Greek_ALPHAACCENT	Greek_kappa	Greek_KAPPA
Greek_epsilonaccent	Greek_EPSILONACCENT	Greek_lamda	Greek_LAMDA
Greek_etaaccent	Greek_ETAACCENT	Greek_lambda	Greek_LAMBDA
Greek_iotaaccent	Greek_IOTAACCENT	Greek_mu	Greek_MU
Greek_iotadieresis	Greek_IOTADIERESIS	Greek_nu	Greek_NU
Greek_omicronaccent	Greek_OMICRONACCENT	Greek_xi	Greek_XI
Greek_upsilonaccent	Greek_UPSILONACCENT	Greek_omicron	Greek_OMICRON
Greek_upsilondieresis	Greek_UPSILONDIERESIS	Greek_pi	Greek_PI
Greek_alpha	Greek_ALPHA	Greek_rho	Greek_RHO
Greek_beta	Greek_BETA	Greek_sigma	Greek_SIGMA
Greek_gamma	Greek_GAMMA	Greek_tau	Greek_TAU
Greek_delta	Greek_DELTA	Greek_upsilon	Greek_UPSILON
Greek_epsilon	Greek_EPSILON	Greek_phi	Greek_PHI
Greek_zeta	Greek_ZETA	Greek_chi	Greek_CHI
Greek_eta	Greek_ETA	Greek_psi	Greek_PSI
Greek_theta	Greek_THETA	Greek_omega	Greek_OMEGA

Capitalization Rules for Other Keysyms

XKB defines no capitalization rules for symbols in any other set of keysyms provided by the consortium. Applications are free to apply additional rules for private keysyms or for other keysyms not covered by XKB.

Appendix B. Canonical Key Types

Canonical Key Types

The ONE_LEVEL Key Type

The *ONE_LEVEL* key type describes groups that have only one symbol. The default *ONE_LEVEL* type has no map entries and does not pay attention to any modifiers.

The TWO_LEVEL Key Type

The *TWO_LEVEL* key type describes groups that have two symbols but are neither alphabetic nor numeric keypad keys. The default *TWO_LEVEL* type uses only the *Shift* modifier. It returns level two if *Shift* is set, level one if it is not.

The ALPHABETIC Key Type

The *ALPHABETIC* key type describes groups that consist of two symbols — the lowercase form of a symbol followed by the uppercase form of the same symbol. The default *ALPHABETIC* type implements locale-sensitive "shift cancels caps lock" behavior using both the *Shift* and *Lock* modifiers as follows:

- If *Shift* and *Lock* are both set, the default *ALPHABETIC* type yields level one.
- If *Shift* alone is set, it yields level two.
- If *Lock* alone is set, it yields level one but preserves the *Lock* modifier.
- If neither *Shift* nor *Lock* are set, it yields level one.

The KEYPAD Key Type

The *KEYPAD* key type describes that consist of two symbols, at least one of which is a numeric keypad symbol. The default *KEYPAD* type implements "shift cancels numeric lock" behavior using the *Shift* modifier and the real modifier bound to the virtual modifier named "NumLock" (the "NumLock" modifier) as follows:

- If *Shift* and the "NumLock" modifier are both set, the default *KEYPAD* type yields level one.
- If either *Shift* or the "NumLock" modifier alone are set, it yields level two.
- If neither *Shift* nor the "NumLock" modifier are set, it yields level one.

Appendix C. New KeySyms

New KeySyms

KeySyms Used by the ISO9995 Standard

Byte 3	**Byte 4**	**Character**	**Name**
254	1		ISO LOCK
254	2		ISO LATCHING LEVEL TWO SHIFT
254	3		ISO LEVEL THREE SHIFT
254	4		ISO LATCHING LEVEL THREE SHIFT
254	5		ISO LEVEL THREE SHIFT LOCK
254	6		ISO LATCHING GROUP SHIFT
254	7		ISO GROUP SHIFT LOCK
254	8		ISO NEXT GROUP
254	9		ISO LOCK NEXT GROUP
254	10		ISO PREVIOUS GROUP
254	11		ISO LOCK PREVIOUS GROUP
254	12		ISO FIRST GROUP
254	13		ISO LOCK FIRST GROUP
254	14		ISO LAST GROUP
254	15		ISO LOCK LAST GROUP
254	32		LEFT TAB
254	33		MOVE LINE UP
254	34		MOVE LINE DOWN
254	35		PARTIAL LINE UP
254	36		PARTIAL LINE DOWN
254	37		PARTIAL SPACE LEFT
254	38		PARTIAL SPACE RIGHT
254	39		SET MARGIN LEFT
254	40		SET MARGIN RIGHT
254	41		RELEASE MARGIN LEFT
254	42		RELEASE MARGIN RIGHT
254	43		RELEASE MARGIN LEFT AND RIGHT
254	44		FAST CURSOR LEFT
254	45		FAST CURSOR RIGHT
254	46		FAST CURSOR UP
254	47		FAST CURSOR DOWN
254	48		CONTINUOUS UNDERLINE

Byte 3	Byte 4	Character	Name
254	49		DISCONTINUOUS UNDERLINE
254	50		EMPHASIZE
254	51		CENTER OBJECT
254	52		ISO_ENTER

KeySyms Used to Control The Core Pointer

Byte 3	Byte 4	Character	Name
254	224		POINTER LEFT
254	225		POINTER RIGHT
254	226		POINTER UP
254	227		POINTER DOWN
254	228		POINTER UP AND LEFT
254	229		POINTER UP AND RIGHT
254	230		POINTER DOWN AND LEFT
254	231		POINTER DOWN AND RIGHT
254	232		DEFAULT POINTER BUTTON
254	233		POINTER BUTTON ONE
254	234		POINTER BUTTON TWO
254	235		POINTER BUTTON THREE
254	236		POINTER BUTTON FOUR
254	237		POINTER BUTTON FIVE
254	238		DEFAULT POINTER BUTTON DOUBLE CLICK
254	239		POINTER BUTTON ONE DOUBLE CLICK
254	240		POINTER BUTTON TWO DOUBLE CLICK
254	241		POINTER BUTTON THREE DOUBLE CLICK
254	242		POINTER BUTTON FOUR DOUBLE CLICK
254	243		POINTER BUTTON FIVE DOUBLE CLICK
254	244		DRAG DEFAULT POINTER BUTTON
254	245		DRAG POINTER BUTTON ONE
254	246		DRAG POINTER BUTTON TWO
254	247		DRAG POINTER BUTTON THREE
254	248		DRAG POINTER BUTTON FOUR
254	249		ENABLE POINTER FROM KEYBOARD
254	250		ENABLE KEYBOARD POINTER ACCEL

Byte 3	Byte 4	Character	Name
254	251		SET DEFAULT POINTER BUTTON NEXT
254	252		SET DEFAULT POINTER BUTTON PREVIOUS
254	253		DRAG POINTER BUTTON FIVE

KeySyms Used to Change Keyboard Controls

Byte 3	Byte 4	Character	Name
254	112		ENABLE ACCESSX KEYS
254	113		ENABLE ACCESSX FEEDBACK
254	114		TOGGLE REPEAT KEYS
254	115		TOGGLE SLOW KEYS
254	116		ENABLE BOUNCE KEYS
254	117		ENABLE STICKY KEYS
254	118		ENABLE MOUSE KEYS
254	119		ENABLE MOUSE KEYS ACCELERATION
254	120		ENABLE OVERLAY1
254	121		ENABLE OVERLAY2
254	122		ENABLE AUDIBLE BELL

KeySyms Used To Control The Server

Byte	Byte	Character	Name
254	208		FIRST SCREEN
254	209		PREVIOUS SCREEN
254	210		NEXT SCREEN
254	211		LAST SCREEN
254	212		TERMINATE SERVER

KeySyms for Non-Spacing Diacritical Keys

Byte	Byte	Character	Name
254	80		DEAD GRAVE ACCENT
254	81		DEAD ACUTE ACCENT
254	82		DEAD CIRCUMFLEX
254	83		DEAD TILDE
254	84		DEAD MACRON
254	85		DEAD BREVE
254	86		DEAD DOT ABOVE
254	87		DEAD DIAERESIS
254	88		DEAD RING ABOVE
254	89		DEAD DOUBLE ACUTE ACCENT
254	90		DEAD CARON
254	91		DEAD CEDILLA
254	92		DEAD OGONEK
254	93		DEAD IOTA
254	94		DEAD VOICED SOUND
254	95		DEAD SEMI VOICED SOUND
254	96		DEAD DOT BELOW

Appendix D. Protocol Encoding

Syntactic Conventions

This document uses the same syntactic conventions as the encoding of the core X protocol, with the following additions:

A LISTofITEMs contains zero or more items of variable type and size. The encode form for a LISTofITEMs is:

```
v     LISTofITEMs           NAME

      TYPE                  MASK-EXPRESSION
      value1                corresponding field(s)
      ...
      valuen                corresponding field(s)
```

The MASK-EXPRESSION is an expression using C-style boolean operators and fields of the request which specifies the bitmask used to determine whether or not a mem ber of the LISTofITEMs is present. If present, TYPE specifies the interpretation of the resulting bitmask and the values are listed using the symbolic names of the members of the set. If TYPE is blank, the values are numeric constants.

It is possible for a single bit in the MASK-EXPRESSION to control more than one ITEM — if the bit is set, all listed ITEMs are present. It is also possible for multiple bits in the MASK-EXPRESSION to control a single ITEM — if any of the bits associated with an ITEM are set, it is present in the LISTofITEMs.

The size of a LISTofITEMS is derived from the items that are present in the list, so it is always given as a variable in the request description, and the request is followed by a section of the form:

```
ITEMs
encode-form
...
encode-form
```

listing an encode-form for each ITEM. The NAME in each encode-form keys to the fields listed as corresponding to each bit in the MASK-EXPRESSION. Items are not necessarily the same size, and the size specified in the encoding form is the size that the item occupies if it is present.

Some types are of variable size. The encode-form for a list of items of a single type but variable size is:

```
S0+..Ss     LISTofTYPE     name
```

Which indicates that the list has *s* elements of variable size and that the size of the list is the sum of the sizes of all of the elements that make up the list. The notation Sn refers to the size of the *n* th element of the list and the notation S* refers to the size of the list as a whole.

The definition of a type of variable size includes an expression which specifies the size. The size is specified as a constant plus a variable expression; the constant specifies the size of the fields that are always present and the variables which make up the variable expression are defined in the constant portion of the structure. For example, the following definition specifies a counted string with a two-byte length field preceding the string:

```
TYPE                 2+n+p
2     n              length
n     STRING8        string
p                    unused,p=pad(n)
```

Some fields are optional. The size of an optional field has the form: "[*expr*]" where expr specifies the size of the field if it is present. An explanation of the conditions under which the field is present follows the name in the encode form:

```
1       BOOL          more
3                     unused
[4]     CARD32        optData, if more==TRUE
```

This portion of the structure is four bytes long if more is FALSE or eight bytes long if more is TRUE. This notation can also be used in size expressions; for example, the size of the previous structure is written as "4+[4]" bytes.

Common Types

```
SETofKB_EVENTTYPE
     #x0001                XkbNewKeyboardNotify
     #x0002                XkbMapNotify
     #x0004                XkbStateNotify
     #x0008                XkbControlsNotify
     #x0010                XkbIndicatorStateNotify
     #x0020                XkbIndicatorMapNotify
     #x0040                XkbNamesNotify
     #x0080                XkbCompatMapNotify
     #x0100                XkbBellNotify
     #x0200                XkbActionMessage
     #x0400                XkbAccessXNotify
     #x0800                XkbExtensionDeviceNotify

SETofKB_NKNDETAIL
     #x01                  XkbNKN_Keycodes
     #x02                  XkbNKN_Geometry
     #x04                  XkbNKN_DeviceID

SETofKB_AXNDETAIL
     #x01                  XkbAXN_SKPress
     #x02                  XkbAXN_SKAccept
     #x04                  XkbAXN_SKReject
     #x08                  XkbAXN_SKRelease
     #x10                  XkbAXN_BKAccept
     #x20                  XkbAXN_BKReject
```

#x40	XkbAXN_AXKWarning

SETofKB_MAPPART

#x0001	XkbKeyTypes
#x0002	XkbKeySyms
#x0004	XkbModifierMap
#x0008	XkbExplicitComponents
#x0010	XkbKeyActions
#x0020	XkbKeyBehaviors
#x0040	XkbVirtualMods
#x0080	XkbVirtualModMap

SETofKB_STATEPART

#x0001	XkbModifierState
#x0002	XkbModifierBase
#x0004	XkbModifierLatch
#x0008	XkbModifierLock
#x0010	XkbGroupState
#x0020	XkbGroupBase
#x0040	XkbGroupLatch
#x0080	XkbGroupLock
#x0100	XkbCompatState
#x0200	XkbGrabMods
#x0400	XkbCompatGrabMods
#x0800	XkbLookupMods
#x1000	XkbCompatLookupMods
#x2000	XkbPointerButtons

SETofKB_BOOLCTRL

#x00000001	XkbRepeatKeys
#x00000002	XkbSlowKeys
#x00000004	XkbBounceKeys
#x00000008	XkbStickyKeys
#x00000010	XkbMouseKeys
#x00000020	XkbMouseKeysAccel
#x00000040	XkbAccessXKeys
#x00000080	XkbAccessXTimeoutMask
#x00000100	XkbAccessXFeedbackMask
#x00000200	XkbAudibleBellMask
#x00000400	XkbOverlay1Mask
#x00000800	XkbOverlay2Mask
#x00001000	XkbIgnoreGroupLockMask

SETofKB_CONTROL

Encodings are the same as for SETofKB_BOOLCTRL, with the addition of:

#x080000000	XkbGroupsWrap
#x100000000	XkbInternalMods
#x200000000	XkbIgnoreLockMods
#x400000000	XkbPerKeyRepeat
#x800000000	XkbControlsEnabled

SETofKB_AXFBOPT

#x0001	XkbAX_SKPressFB
#x0002	XkbAX_SKAcceptFB
#x0004	XkbAX_FeatureFB

```
     #x0008                  XkbAX_SlowWarnFB
     #x0010                  XkbAX_IndicatorFB
     #x0020                  XkbAX_StickyKeysFB
     #x0100                  XkbAX_SKReleaseFB
     #x0200                  XkbAX_SKRejectFB
     #x0400                  XkbAX_BKRejectFB
     #x0800                  XkbAX_DumbBell

SETofKB_AXSKOPT
     #x0040                  XkbAX_TwoKeys
     #x0080                  XkbAX_LatchToLock

SETofKB_AXOPTION
     Encoding same as the bitwise union of :
     SETofKB_AXFBOPT
     SETofKB_AXSKOPT

KB_DEVICESPEC
     0..255     input extension device id
     #x100      XkbUseCoreKbd
     #x200      XkbUseCorePtr

KB_LEDCLASSRESULT
     0     KbdFeedbackClass
     4     LedFeedbackClass

KB_LEDCLASSSPEC
     Encoding same as KB_LEDCLASSRESULT, with the addition of:
     #x0300     XkbDfltXIClass
     #x0500     XkbAllXIClasses

KB_BELLCLASSRESULT
     0     KbdFeedbackClass
     5     BellFeedbackClass

KB_BELLCLASSSPEC
     Encoding same as KB_BELLCLASSRESULT, with the addition of:
     #x0300     XkbDfltXIClass

KB_IDSPEC
     0..255     input extension feedback id
     #x0400     XkbDfltXIId

KB_IDRESULT
     Encoding same as KB_IDSPEC, with the addition of:
     #xff00     XkbXINone

KB_MULTIIDSPEC
     encodings same as KB_IDSPEC, with the addition of:
     #x0500     XkbAllXIIds

KB_GROUP
     0     XkbGroup1
     1     XkbGroup2
     2     XkbGroup3
     3     XkbGroup4
```

```
KB_GROUPS
     Encoding same as KB_GROUP, with the addition of:
     254     XkbAnyGroup
     255     XkbAllGroups

SETofKB_GROUP
     #x01     XkbGroup1
     #x02     XkbGroup2
     #x04     XkbGroup3
     #x08     XkbGroup4

SETofKB_GROUPS
     Encoding same as SETofKB_GROUP, with the addition of:
     #x80     XkbAnyGroup

KB_GROUPSWRAP
     #x00     XkbWrapIntoRange
     #x40     XkbClampIntoRange
     #x80     XkbRedirectIntoRange

SETofKB_VMODSHIGH
     #x80     virtual modifier 15
     #x40     virtual modifier 14
     #x20     virtual modifier 13
     #x10     virtual modifier 12
     #x08     virtual modifier 11
     #x04     virtual modifier 10
     #x02     virtual modifier 9
     #x01     virtual modifier 8

SETofKB_VMODSLOW
     #x80     virtual modifier 7
     #x40     virtual modifier 6
     #x20     virtual modifier 5
     #x10     virtual modifier 4
     #x08     virtual modifier 3
     #x04     virtual modifier 2
     #x02     virtual modifier 1
     #x01     virtual modifier 0

SETofKB_VMOD
     #x8000     virtual modifier 15
     #x4000     virtual modifier 14
     #x2000     virtual modifier 13
     #x1000     virtual modifier 12
     #x0800     virtual modifier 11
     #x0400     virtual modifier 10
     #x0200     virtual modifier 9
     #x0100     virtual modifier 8
     #x0080     virtual modifier 7
     #x0040     virtual modifier 6
     #x0020     virtual modifier 5
     #x0010     virtual modifier 4
     #x0008     virtual modifier 3
     #x0004     virtual modifier 2
```

```
     #x0002     virtual modifier 1
     #x0001     virtual modifier 0

SETofKB_EXPLICIT
     #x80     XkbExplicitVModMap
     #x40     XkbExplicitBehavior
     #x20     XkbExplicitAutoRepeat
     #x10     XkbExplicitInterpret
     #x08     XkbExplicitKeyType4
     #x04     XkbExplicitKeyType3
     #x02     XkbExplicitKeyType2
     #x01     XkbExplicitKeyType1

KB_SYMINTERPMATCH
     #x80     XkbSI_LevelOneOnly
     #x7f     operation, one of the following:
          0 XkbSI_NoneOf
          1 XkbSI_AnyOfOrNone
          2 XkbSI_AnyOf
          3 XkbSI_AllOf
          4 XkbSI_Exactly

SETofKB_IMFLAG
     #x80     XkbIM_NoExplicit
     #x40     XkbIM_NoAutomatic
     #x20     XkbIM_LEDDrivesKB

SETofKB_IMMODSWHICH
     #x10     XkbIM_UseCompat
     #x08     XkbIM_UseEffective
     #x04     XkbIM_UseLocked
     #x02     XkbIM_UseLatched
     #x01     XkbIM_UseBase

SETofKB_IMGROUPSWHICH
     #x10     XkbIM_UseCompat
     #x08     XkbIM_UseEffective
     #x04     XkbIM_UseLocked
     #x02     XkbIM_UseLatched
     #x01     XkbIM_UseBase

KB_INDICATORMAP
1     SETofKB_IMFLAGS          flags
1     SETofKB_IMGROUPSWHICH          whichGroups
1     SETofKB_GROUP          groups
1     SETofKB_IMMODSWHICH          whichMods
1     SETofKEYMASK          mods
1     SETofKEYMASK          realMods
2     SETofKB_VMOD          vmods
4     SETofKB_BOOLCTRL          ctrls

SETofKB_CMDETAIL
     #x01     XkbSymInterp
     #x02     XkbGroupCompat

SETofKB_NAMEDETAIL
```

#x0001	XkbKeycodesName
#x0002	XkbGeometryName
#x0004	XkbSymbolsName
#x0008	XkbPhysSymbolsName
#x0010	XkbTypesName
#x0020	XkbCompatName
#x0040	XkbKeyTypeNames
#x0080	XkbKTLevelNames
#x0100	XkbIndicatorNames
#x0200	XkbKeyNames
#x0400	XkbKeyAliases
#x0800	XkbVirtualModNames
#x1000	XkbGroupNames
#x2000	XkbRGNames

SETofKB_GBNDETAIL

#x01	XkbGBN_Types
#x02	XkbGBN_CompatMap
#x04	XkbGBN_ClientSymbols
#x08	XkbGBN_ServerSymbols
#x10	XkbGBN_IndicatorMaps
#x20	XkbGBN_KeyNames
#x40	XkbGBN_Geometry
#x80	XkbGBN_OtherNames

SETofKB_XIEXTDEVFEATURE

#x02	XkbXI_ButtonActions
#x04	XkbXI_IndicatorNames
#x08	XkbXI_IndicatorMaps
#x10	XkbXI_IndicatorState

SETofKB_XIFEATURE

Encoding same as SETofKB_XIEXTDEVFEATURE, with the addition of:

#x01	XkbXI_Keyboards

SETofKB_XIDETAIL

Encoding same as SETofKB_XIFEATURE, with the addition of:

#x8000	XkbXI_UnsupportedFeature

SETofKB_PERCLIENTFLAG

#x01	XkbDetectableAutorepeat
#x02	XkbGrabsUseXKBState
#x04	XkbAutoResetControls
#x08	XkbLookupStateWhenGrabbed
#x10	XkbSendEventUsesXKBState

KB_MODDEF

1	SETofKEYMASK	mask
1	SETofKEYMASK	realMods
2	SETofVMOD	vmods

KB_COUNTED_STRING8

1	l	length
l	STRING8	string

KB_COUNTED_STRING16

```
2     l          length
l     STRING8          string

KB_COUNTED_STRING16
p               unused,p=pad(2+l)
```

Errors

```
1     0          Error
2     ??          code
2     CARD16          sequence
4     CARD32          error value
     most significant 8 bits of error value have the meaning:
     0xff     XkbErrBadDevice
     0xfe     XkbErrBadClass
     0xfd     XkbErrBadId
     the least significant 8 bits of the error value contain the device id,
class, or feedback
     id which failed.
2     CARD16          minor opcode
1     CARD8          major opcode
21               unused
```

Key Actions

```
1     0          type
7               unused

1     1          type
1     BITMASK          flags
     #x01     XkbSA_ClearLocks
     #x02     XkbSA_LatchToLock
     #x04     XkbSA_UseModMapMods
1     SETofKEYMASK          mask
1     SETofKEYMASK          real modifiers
1     SETofKB_VMODSHIGH          virtual modifiers high
1     SETofKB_VMODSLOW          virtual modifiers low
2               unused

1     2          type
1     BITMASK          flags
     #x01     XkbSA_ClearLocks
     #x02     XkbSA_LatchToLock
     #x04     XkbSA_UseModMapMods
1     SETofKEYMASK          mask
1     SETofKEYMASK          real modifiers
1     SETofKB_VMODSHIGH          virtual modifiers high
1     SETofKB_VMODSLOW          virtual modifiers low
2               unused

1     3          type
1     BITMASK          flags
     #x01     XkbSA_LockNoLock
     #x02     XkbSA_LockNoUnlock
```

	#x04	XkbSA_UseModMapMods
1	SETofKEYMASK	mask
1	SETofKEYMASK	real modifiers
1	SETofKB_VMODSHIGH	virtual modifiers high
1	SETofKB_VMODSLOW	virtual modifiers low
2		unused

1	4	type
1	BITMASK	flags
	#x01	XkbSA_ClearLocks
	#x02	XkbSA_LatchToLock
	#x04	XkbSA_GroupAbsolute
1	INT8	group
5		unused

1	5	type
1	BITMASK	flags
	#x01	XkbSA_ClearLocks
	#x02	XkbSA_LatchToLock
	#x04	XkbSA_GroupAbsolute
1	INT8	group
5		unused

1	6	type
1	BITMASK	flags
	#x01	XkbSA_LockNoLock
	#x02	XkbSA_LockNoUnlock
	#x04	XkbSA_GroupAbsolute
1	INT8	group
5		unused

1	7	type
1	BITMASK	flags
	#x01	XkbSA_NoAcceleration
	#x02	XkbSA_MoveAbsoluteX
	#x04	XkbSA_MoveAbsoluteY
1	INT8	x high
1	CARD8	x low
1	INT8	y high
1	CARD8	y low
2		unused

1	8	type
1	BITMASK	flags
1	CARD8	count
1	CARD8	button
4		unused

1	9	type
1	BITMASK	flags
1		unused
1	CARD8	button
4		unused

1	10	type

```
1     BITMASK          flags
     #x02     XkbSA_DfltBtnAbsolute
1     BITMASK          affect
     #x01     XkbSA_AffectDfltBtn
1     INT8          value
4              unused

1     11          type
1     BITMASK          flags
     #x01     XkbSA_LockNoLock
     #x02     XkbSA_LockNoUnlock
     #x04     XkbSA_UseModMapMods (if SA_ISODfltIsGroup is 0)
     #x04     XkbSA_GroupAbsolute (if SA_ISODfltIsGroup is 1)
     #x80     XkbSA_ISODfltIsGroup
1     SETofKEYMASK          mask
1     SETofKEYMASK          real modifiers
1     INT8          group
1     BITMASK          affect
     #x08     XkbSA_ISONoAffectCtrls
     #x10     XkbSA_ISONoAffectPtr
     #x20     XkbSA_ISONoAffectGroup
     #x40     XkbSA_ISONoAffectMods
1     SETofKB_VMODSHIGH          virtual modifiers high
1     SETofKB_VMODSLOW          virtual modifiers low

1     12          type
7              unused

1     13          type
1     BITMASK          flags
     #x01     XkbSA_SwitchApplication
     #x04     XkbSA_SwitchAbsolute
1     INT8          new screen
5              unused (must be 0)

1     14          type
3              unused (must be 0)
1     BITMASK          boolean controls high
     #x01     XkbAccessXFeedbackMask
     #x02     XkbAudibleBellMask
     #x04     XkbOverlay1Mask
     #x08     XkbOverlay2Mask
     #x10     XkbIgnoreGroupLockMask
1     BITMASK          boolean controls low
     #x01     XkbRepeatKeys
     #x02     XkbSlowKeys
     #x04     XkbBounceKeys
     #x08     XkbStickyKeys
     #x10     XkbMouseKeys
     #x20     XkbMouseKeysAccel
     #x40     XkbAccessXKeys
     #x80     XkbAccessXTimeoutMask
2          unused (must be 0)

1     15          type
```

```
3               unused (must be 0)
1     BITMASK          boolean controls high
     #x01     XkbAccessXFeedbackMask
     #x02     XkbAudibleBellMask
     #x04     XkbOverlay1Mask
     #x08     XkbOverlay2Mask
     #x10     XkbIgnoreGroupLockMask
1     BITMASK          boolean controls low
     #x01     XkbRepeatKeys
     #x02     XkbSlowKeys
     #x04     XkbBounceKeys
     #x08     XkbStickyKeys
     #x10     XkbMouseKeys
     #x20     XkbMouseKeysAccel
     #x40     XkbAccessXKeys
     #x80     XkbAccessXTimeoutMask

1     16          type
2               unused (must be 0)

1     16          type
1     BITMASK          flags
     #x01          XkbSA_MessageOnPress
     #x02          XkbSA_MessageOnRelease
     #x04          XkbSA_MessageGenKeyEvent
6     STRING           message

1     17          type
1     KEYCODE          new key
1     SETofKEYMASK          mask
1     SETofKEYMASK          real modifiers
1     SETofKB_VMODSHIGH          virtual modfiiers mask high
1     SETofKB_VMODSLOW          virtual modifiers mask low
1     SETofKB_VMODSHIGH          virtual modifiers high
1     SETofKB_VMODSLOW          virtual modfiers low

1     18          type
1     0          flags
1     CARD8          count
1     CARD8          button
1     CARD8          device
3               unused (must be 0)

1     19          type
1     BITMASK          flags
     #x01          XkbSA_LockNoLock
     #x02          XkbSA_LockNoUnlock
1               unused
1     CARD8          button
1     CARD8          device

1     20          type
1     CARD8          device
1     KB_SA_VALWHAT          valuator 1 what
     #x00          XkbSA_IgnoreVal
```

```
     #x01          XkbSA_SetValMin
     #x02          XkbSA_SetValCenter
     #x03          XkbSA_SetValMax
     #x04          XkbSA_SetValRelative
     #x05          XkbSA_SetValAbsolute
1     CARD8          valuator 1 index
1     CARD8          valuator 1 value
1     KB_SA_VALWHAT          valuator 2 what
     Encodings as for "valuator 1 what" above
1     CARD8          valuator 2 index
1     CARD8          valuator 2 value
```

Key Behaviors

```
1     #x00          type
1               unused

1     #x01          type
1               unused

1     #x02          type
1     0..31          group

1     #x03          type
1     KEYCODE          key

1     #x04          type
1     CARD8          key

1     #x81          type
1               unused

1     #x82          type
1     0..31          group

1     #x83          type
1     KEYCODE          key

1     #x84          type
1     KEYCODE          key
```

Requests

```
1     ??          opcode
1     0          xkb-opcode
2     2          request-length
2     CARD16          wantedMajor
2     CARD16          wantedMinor

1     1          Reply
1     BOOL          supported
2     CARD16          sequence number
4     0          reply length
2     1          serverMajor
2     0          serverMinor
```

```
20              unused

1     ??          opcode
1     1          xkb-opcode
2     4+(V+p)/4          request-length
2     KB_DEVICESPEC           deviceSpec
2     SETofKB_EVENTTYPE          affectWhich
2     SETofKB_EVENTTYPE          clear
2     SETofKB_EVENTTYPE          selectAll
2     SETofKB_MAPDETAILS          affectMap
2     SETofKB_MAPDETAILS          map
V     LISTofITEMs          details
     SETofKB_EVENTTYPE
(affectWhich&(~clear)&(~selectAll))
     XkbNewKeyboardNotify     affectNewKeyboard, newKeyboardDetails
     XkbStateNotify     affectState, stateDetails
     XkbControlsNotify     affectCtrls, ctrlDetails
     XkbIndicatorStateNotify     affectIndicatorState, indicatorStateDetails
     XkbIndicatorMapNotify     affectIndicatorMap, indicatorMapDetails
     XkbNamesNotify     affectNames, namesDetails
     XkbCompatMapNotify     affectCompat, compatDetails
     XkbBellNotify     affectBell, bellDetails
     XkbActionMessage     affectMsgDetails, msgDetails
     XkbExtensionDeviceNotify     affectExtDev, extdevDetails

ITEMs
p          unused, p=pad(V)

ITEMs
2     SETofKB_NKNDETAIL          affectNewKeyboard
2     SETofKB_NKNDETAIL          newKeyboardDetails
2     SETofKB_STATEPART          affectState
2     SETofKB_STATEPART          stateDetails
4     SETofKB_CONTROL          affectCtrls
4     SETofKB_CONTROL          ctrlDetails
4     SETofKB_INDICATOR          affectIndicatorState
4     SETofKB_INDICATOR          indicatorStateDetails
4     SETofKB_INDICATOR          affectIndicatorMaps
4     SETofKB_INDICATOR          indicatorMapDetails
2     SETofKB_NAME_DETAIL          affectNames
2     SETofKB_NAME_DETAIL          namesDetails
1     SETofKB_CMDETAIL          affectCompat
1     SETofKB_CMDETAIL          compatDetails
1     SETofKB_BELLDETAIL          affectBell
1     SETofKB_BELLDETAIL          bellDetails
1     SETofKB_MSGDETAIL          affectMsgDetails
1     SETofKB_MSGDETAIL          msgDetails
2     SETofKB_AXNDETAIL          affectAccessX
2     SETofKB_AXNDETAIL          accessXDetails
2     SETofKB_XIDETAIL          affectExtDev
2     SETofKB_XIDETAIL          extdevDetails

1     ??          opcode
1     3          xkb-opcode
2     7          request-length
```

2	KB_DEVICESPEC	deviceSpec
2	KB_BELLCLASSSPEC	bellClass
2	KB_IDSPEC	bellID
1	INT8	percent
1	BOOL	forceSound
1	BOOL	eventOnly
1		unused
2	INT16	pitch
2	INT16	duration
2		unused
4	ATOM	name
4	WINDOW	window

1	??	opcode
1	4	xkb-opcode
2	2	request-length
2	KB_DEVICESPEC	deviceSpec
2		unused

1	1	Reply
1	CARD8	deviceID
2	CARD16	sequence number
4	0	length
1	SETofKEYMASK	mods
1	SETofKEYMASK	baseMods
1	SETofKEYMASK	latchedMods
1	SETofKEYMASK	lockedMods
1	KP_GROUP	group
1	KP_GROUP	lockedGroup
2	INT16	baseGroup
2	INT16	latchedGroup
1	SETofKEYMASK	compatState
1	SETofKEYMASK	grabMods
1	SETofKEYMASK	compatGrabMods
1	SETofKEYMASK	lookupMods
1	SETofKEYMASK	compatLookupMods
1		unused
2	SETofBUTMASK	ptrBtnState
6		unused

1	??	opcode
1	5	xkb-opcode
2	4	request-length
2	KB_DEVICESPEC	deviceSpec
1	SETofKEYMASK	affectModLocks
1	SETofKEYMASK	modLocks
1	BOOL	lockGroup
1	KB_GROUP	groupLock
1	SETofKEYMASK	affectModLatches
1	SETofKEYMASK	modLatches
1		unused
1	BOOL	latchGroup
2	INT16	groupLatch

1	??	opcode

```
1     6          xkb-opcode
2     2          request-length
2     KB_DEVICESPEC          deviceSpec
2               unused

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     15          length
1     CARD8          mouseKeysDfltBtn
1     CARD8          numGroups
1     CARD8          groupsWrap
1     SETofKEYMASK          internalMods.mask
1     SETofKEYMASK          ignoreLockMods.mask
1     SETofKEYMASK          internalMods.realMods
1     SETofKEYMASK          ignoreLockMods.realMods
1               unused
2     SETofKB_VMOD          internalMods.vmods
2     SETofKB_VMOD          ignoreLockMods.vmods
2     CARD16          repeatDelay
2     CARD16          repeatInterval
2     CARD16          slowKeysDelay
2     CARD16          debounceDelay
2     CARD16          mouseKeysDelay
2     CARD16          mouseKeysInterval
2     CARD16          mouseKeysTimeToMax
2     CARD16          mouseKeysMaxSpeed
2     INT16          mouseKeysCurve
2     SETofKB_AXOPTION          accessXOptions
2     CARD16          accessXTimeout
2     SETofKB_AXOPTION          accessXTimeoutOptionsMask
2     SETofKB_AXOPTION          accessXTimeoutOptionValues
2               unused
4     SETofKB_BOOLCTRL          accessXTimeoutMask
4     SETofKB_BOOLCTRL          accessXTimeoutValues
4     SETofKB_BOOLCTRL          enabledControls
32     LISTofCARD8          perKeyRepeat

1     ??          opcode
1     7          xkb-opcode
2     25          request-length
2     KB_DEVICESPEC          deviceSpec
1     SETofKEYMASK          affectInternalRealMods
1     SETofKEYMASK          internalRealMods
1     SETofKEYMASK          affectIgnoreLockRealMods
1     SETofKEYMASK          ignoreLockRealMods
2     SETofKB_VMOD          affectInternalVirtualMods
2     SETofKB_VMOD          internalVirtualMods
2     SETofKB_VMOD          affectIgnoreLockVirtualMods
2     SETofKB_VMOD          ignoreLockVirtualMods
1     CARD8          mouseKeysDfltBtn
1     CARD8          groupsWrap
2     SETofKB_AXOPTION          accessXOptions
2               unused
```

```
4     SETofKB_BOOLCTRL           affectEnabledControls
4     SETofKB_BOOLCTRL           enabledControls
4     SETofKB_CONTROL          changeControls
2     CARD16          repeatDelay
2     CARD16          repeatInterval
2     CARD16          slowKeysDelay
2     CARD16          debounceDelay
2     CARD16          mouseKeysDelay
2     CARD16          mouseKeysInterval
2     CARD16          mouseKeysTimeToMax
2     CARD16          mouseKeysMaxSpeed
2     INT16          mouseKeysCurve
2     CARD16          accessXTimeout
4     SETofKB_BOOLCTRL           accessXTimeoutMask
4     SETofKB_BOOLCTRL           accessXTimeoutValues
2     SETofKB_AXOPTION          accessXTimeoutOptionsMask
2     SETofKB_AXOPTION          accessXTimeoutOptionsValues
32     LISTofCARD8          perKeyRepeat

1     CARD8          opcode
1     8          xkb-opcode
2     7          request-length
2     KB_DEVICESPEC           deviceSpec
2     SETofKB_MAPPART          full
2     SETofKB_MAPPART          partial
1     CARD8          firstType
1     CARD8          nTypes
1     KEYCODE          firstKeySym
1     CARD8          nKeySyms
1     KEYCODE          firstKeyAction
1     CARD8          nKeyActions
1     KEYCODE          firstKeyBehavior
1     CARD8          nKeyBehaviors
2     SETofKB_VMOD          virtualMods
1     KEYCODE          firstKeyExplicit
1     CARD8          nKeyExplicit
1     KEYCODE          firstModMapKey
1     CARD8          nModMapKeys
1     KEYCODE          firstVModMapKey
1     CARD8          nVModMapKeys
2               unused

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     2+(I/4)          length
2               unused
1     KEYCODE          minKeyCode
1     KEYCODE          maxKeyCode
2     SETofKB_MAPPART          present
1     CARD8          firstType
1     t          nTypes
1     CARD8          totalTypes
1     KEYCODE          firstKeySym
```

```
2      S           totalSyms
1      s           nKeySyms
1      KEYCODE           firstKeyAction
2      A           totalActions
1      a           nKeyActions
1      KEYCODE           firstKeyBehavior
1      b           nKeyBehaviors
1      B           totalKeyBehaviors
1      KEYCODE           firstKeyExplicit
1      e           nKeyExplicit
1      E           totalKeyExplicit
1      KEYCODE           firstModMapKey
1      m           nModMapKeys
1      M           totalModMapKeys
1      KEYCODE           firstVModMapKey
1      0           nVModMapKeys
1      V           totalVModMapKeys
1                unused
2      SETofKB_VMOD           virtualMods (has v bits set to 1)
I      LISTofITEMs           map
     SETofKB_MAPPART                 (present)
     XkbKeyTypes     typesRtrn
     XkbKeySyms     symsRtrn
     XkbKeyActions     actsRtrn.count, actsRtrn.acts
     XkbKeyBehaviors     behaviorsRtrn
     XkbVirtualMods     vmodsRtrn
     XkbExplicitComponents     explicitRtrn
     XkbModifierMap     modmapRtrn
     XkbVirtualModMap     vmodMapRtrn

ITEMs
T1+..Tt     LISTofKB_KEYTYPE           typesRtrn
8s+4S     LISTofKB_KEYSYMMAP           symsRtrn
a      LISTofCARD8           actsRtrn.count
p                unused,p=pad(a)
8A      LISTofKB_ACTION           actsRtrn.acts
4B      LISTofKB_SETBEHAVIOR           behaviorsRtrn
v      LISTofSETofKEYMASK           vmodsRtrn
p                unused, p=pad(v)
2E      LISTofKB_SETEXPLICIT           explicitRtrn
p                unused,p=pad(2E)
2M      LISTofKB_KEYMODMAP           modmapRtrn
p                unused, p=pad(2M)
4V      LISTofKB_KEYVMODMAP           vmodMapRtrn

KB_KEYTYPE     8+8m+[4m]
1      SETofKEYMASK           mods.mask
1      SETofKEYMASK           mods.mods
2      SETofKB_VMOD           mods.vmods
1      CARD8           numLevels
1      m           nMapEntries
1      BOOL           hasPreserve
1                unused
8m      LISTofKB_KTMAPENTRY           map
```

```
[4m]      LISTofKB_MODDEF           preserve

KB_KTMAPENTRY
1     BOOL          active
1     SETofKEYMASK          mods.mask
1     CARD8          level
1     SETofKEYMASK          mods.mods
2     SETofKB_VMOD          mods.vmods
2               unused

KB_KEYSYMMAP     8+4n
4     LISTofCARD8          ktIndex
1     CARD8          groupInfo
1     CARD8          width
2     n          nSyms
4n     LISTofKEYSYM          syms

KB_SETBEHAVIOR
1     KEYCODE          keycode
2     KB_BEHAVIOR          behavior
1               unused

KB_SETEXPLICIT
1     KEYCODE          keycode
1     SETofKB_EXPLICIT          explicit

KB_KEYMODMAP
1     KEYCODE          keycode
1     SETofKB_KEYMASK          mods

KB_KEYVMODMAP
1     KEYCODE          keycode
1               unused
2     SETofKB_VMOD          vmods

1     CARD8          opcode
1     9          xkb-opcode
2     9+(I/4)          request-length
2     KB_DEVICESPEC          deviceSpec
2     SETofKB_MAPPART          present
2     SETofKB_SETMAPFLAGS          flags
     #0001     SetMapResizeTypes
     #0002     SetMapRecomputeActions
1     KEYCODE          minKeyCode
1     KEYCODE          maxKeyCode
1     CARD8          firstType
1     t          nTypes
1     KEYCODE          firstKeySym
1     s          nKeySyms
2     S          totalSyms
1     KEYCODE          firstKeyAction
1     a          nKeyActions
2     A          totalActions
1     KEYCODE          firstKeyBehavior
1     b          nKeyBehaviors
```

```
1     B          totalKeyBehaviors
1     KEYCODE          firstKeyExplicit
1     e          nKeyExplicit
1     E          totalKeyExplicit
1     KEYCODE          firstModMapKey
1     m          nModMapKeys
1     M          totalModMapKeys
1     KEYCODE          firstVModMapKey
1     v          nVModMapKeys
1     V          totalVModMapKeys
2     SETofKB_VMOD          virtualMods (has n bits set to 1)
I     LISTofITEMs          values
     SETofKB_MAPPART          (present)
     XkbKeyTypes     types
     XkbKeySymbols     syms
     XkbKeyActions     actions.count,actions.actions
     XkbKeyBehaviors     behaviors
     XkbVirtualMods     vmods
     XkbExplicitComponents     explicit
     XkbModifierMap     modmap
     XkbVirtualModMap     vmodmap

ITEMs
T0+..Tt     LISTofKB_SETKEYTYPE          types
8s+4S     LISTofKB_KEYSYMMAP          syms
a     LISTofCARD8          actions.count
p               unused,p=pad(a)
8A     LISTofKB_ACTION          actions.actions
4B     LISTofKB_SETBEHAVIOR          behaviors
v     LISTofSETofKEYMASK          vmods
p               unused, p=pad(v)
2E     LISTofKB_SETEXPLICIT          explicit
p               unused,p=pad(2E)
2M     LISTofKB_KEYMODMAP          modmap
P               unused, p=pad(2M)
4V     LISTofKB_KEYVMODMAP          vmodmap

KB_SETKEYTYPE     8+4m+[4m]
1     SETofKEYMASK          mask
1     SETofKEYMASK          realMods
2     SETofKB_VMOD          virtualMods
1     CARD8          numLevels
1     m          nMapEntries
1     BOOL          preserve
1               unused
4m     LISTofKB_KTSETMAPENTRY          entries
[4m]     LISTofKB_MODDEF          preserveEntries (if preserve==TRUE)

KB_KTSETMAPENTRY
1     CARD8          level
1     SETofKEYMASK          realMods
2     SETofKB_VMOD          virtualMods

1     ??          opcode
1     10          xkb-opcode
```

```
2     3          request-length
2     KB_DEVICESPEC           deviceSpec
1     SETofKB_GROUP          groups
1     BOOL          getAllSI
2     CARD16          firstSI
2     CARD16          nSI

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     (16n+4g)/4          length
1     SETofKB_GROUP          groupsRtrn (has g bits set to 1)
1               unused
2     CARD16          firstSIRtrn
2     n          nSIRtrn
2     CARD16          nTotalSI
16               unused
16n     LISTofKB_SYMINTERPRET          siRtrn
4g     LISTofKB_MODDEF          groupRtrn

1     ??          opcode
1     11          xkb-opcode
2     4+(16n+4g)          request-length
2     KB_DEVICESPEC           deviceSpec
1               unused
1     BOOL          recomputeActions
1     BOOL          truncateSI
1     SETofKB_GROUP          groups (has g bits set to 1)
2     CARD16          firstSI
2     n          nSI
2               unused
16n     LISTofKB_SYMINTERPRET          si
4g     LISTofKB_MODDEF          groupMaps

1     ??          opcode
1     12          xkb-opcode
2     2          request-length
2     KB_DEVICESPEC           deviceSpec

ITEMs
2               unused

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     0          length
4     SETofKB_INDICATOR          state
20               unused

1     ??          opcode
1     13          xkb-opcode
2     3          request-length
2     KB_DEVICESPEC           deviceSpec
2               unused
4     SETofKB_INDICATOR          which
```

1	1	Reply
1	CARD8	deviceID
2	CARD16	sequence number
4	12n/4	length
4	SETofKB_INDICATOR	which (has n bits set to 1)
4	SETofKB_INDICATOR	realIndicators
1	n	nIndicators
15		unused
12n	LISTofKB_INDICATORMAP	maps

1	??	opcode
1	14	xkb-opcode
2	3+3n	request-length
2	KB_DEVICESPEC	deviceSpec
2		unused
4	SETofKB_INDICATOR	which (has n bits set to 1)
12n	LISTofKB_INDICATORMAP	maps

1	CARD8	opcode
1	15	xkb-opcode
2	4	request-length
2	KB_DEVICESPEC	deviceSpec
2	KB_LEDCLASSSPEC	ledClass
2	KB_IDSPEC	ledID
2		unused
4	ATOM	indicator

1	1	Reply
1	CARD8	deviceID
2	CARD16	sequence number
4	0	length
4	ATOM	indicator
1	BOOL	found
1	BOOL	on
1	BOOL	realIndicator
1	KB_INDICATOR	ndx
1	SETofKB_IMFLAGS	map.flags
1	SETofKB_IMGROUPSWHICH	map.whichGroups
1	SETofKB_GROUPS	map.groups
1	SETofKB_IMMODSWHICH	map.whichMods
1	SETofKEYMASK	map.mods
1	SETofKEYMASK	map.realMods
2	SETofKB_VMOD	map.vmods
4	SETofKB_BOOLCTRL	map.ctrls
1	BOOL	supported
3		unused

1	??	opcode
1	16	xkb-opcode
2	8	request-length
2	KB_DEVICESPEC	deviceSpec
2	KB_LEDCLASSSPEC	ledClass
2	KB_IDSPEC	ledID
2		unused
4	ATOM	indicator

```
1     BOOL          setState
1     BOOL          on
1     BOOL          setMap
1     BOOL          createMap
1               unused
1     SETofKB_IMFLAGS          map.flags
1     SETofKB_IMGROUPSWHICH          map.whichGroups
1     SETofKB_GROUP          map.groups
1     SETofKB_IMMODSWHICH          map.whichMods
1     SETofKEYMASK          map.realMods
2     SETofKB_VMOD          map.vmods
4     SETofKB_BOOLCTRL          map.ctrls

1     CARD8          opcode
1     17          xkb-opcode
2     3          request-length
2     KB_DEVICESPEC          deviceSpec
2               unused
4     SETofKB_NAMEDETAIL          which

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     V/4          length
4     SETofKB_NAMEDETAIL          which
1     KEYCODE          minKeyCode
1     KEYCODE          maxKeyCode
1     t          nTypes
1     SETofKB_GROUP          groupNames (has g bits set to 1)
2     SETofKB_VMOD          virtualMods (has v bits set to 1)
1     KEYCODE          firstKey
1     k          nKeys
4     SETofKB_INDICATOR          indicators (has i bits set to 1)
1     r          nRadioGroups
1     a          nKeyAliases
2     l          nKTLevels
4               unused
V     LISTofITEMs          valueList
     SETofKB_NAMEDETAIL          (which)
     XkbKeycodesName     keycodesName
     XkbGeometryName     geometryName
     XkbSymbolsName     symbolsName
     XkbPhySymbolsName     physSymbolsName
     XkbTypesName     typesName
     XkbCompatName     compatName
     XkbKeyTypeNames     typeNames
     XkbKTLevelNames     nLevelsPerType, ktLevelNames
     XkbIndicatorNames     indicatorNames
     XkbVirtualModNames     virtualModNames
     XkbGroupNames     groupNames
     XkbKeyNames     keyNames
     XkbKeyAliases     keyAliases
     XkbRGNames     radioGroupNames

ITEMs
```

```
4     ATOM          keycodesName
4     ATOM          geometryName
4     ATOM          symbolsName
4     ATOM          physSymbolsName
4     ATOM          typesName
4     ATOM          compatName
4t     LISTofATOM          typeNames
l     LISTofCARD8          nLevelsPerType, sum of all elements=L
p               unused, p=pad(l)
4L     LISTofATOM          ktLevelNames
4i     LISTofATOM          indicatorNames
4v     LISTofATOM          virtualModNames
4g     LISTofATOM          groupNames
4k     LISTofKB_KEYNAME          keyNames
8a     LISTofKB_KEYALIAS          keyAliases
4r     LISTofATOM          radioGroupNames

1     CARD8          opcode
1     18          xkb-opcode
2     7+(V/4)          request-length
2     KB_DEVICESPEC          deviceSpec
2     SETofKB_VMOD          virtualMods
4     SETofKB_NAMEDETAIL          which
1     CARD8          firstType
1     t          nTypes
1     CARD8          firstKTLevel
1     l          nKTLevels
4     SETofKB_INDICATOR          indicators (has i bits set to 1)
1     SETofKB_GROUP          groupNames (has g bits set to 1)
1     r          nRadioGroups
1     KEYCODE          firstKey
1     k          nKeys
1     a          nKeyAliases
1               unused
2     L          totalKTLevelNames
V     LISTofITEMs          values
     SETofKB_NAMEDETAIL          (which)
     XkbKeycodesName     keycodesName
     XkbGeometryName     geometryName
     XkbSymbolsName     symbolsName
     XkbPhySymbolsName     physSymbolsName
     XkbTypesName     typesName
     XkbCompatName     compatName
     XkbKeyTypeNames     typeNames
     XkbKTLevelNames     nLevelsPerType, ktLevelNames
     XkbIndicatorNames     indicatorNames
     XkbVirtualModNames     virtualModNames
     XkbGroupNames     groupNames
     XkbKeyNames     keyNames
     XkbKeyAliases     keyAliases
     XkbRGNames     radioGroupNames

ITEMs
4     ATOM          keycodesName
```

```
4     ATOM          geometryName
4     ATOM          symbolsName
4     ATOM          physSymbolsName
4     ATOM          typesName
4     ATOM          compatName
4t     LISTofATOM          typeNames
l     LISTofCARD8          nLevelsPerType
p               unused, p=pad(l)
4L     LISTofATOM          ktLevelNames
4i     LISTofATOM          indicatorNames
4v     LISTofATOM          virtualModNames
4g     LISTofATOM          groupNames
4k     LISTofKB_KEYNAME          keyNames
8a     LISTofKB_KEYALIAS          keyAliases
4r     LISTofATOM          radioGroupNames

1     CARD8          opcode
1     19          xkb-opcode
2     3          request-length
2     KB_DEVICESPEC          deviceSpec
2               unused
4     ATOM          name

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     (f+8p+C*+H*+S*+D*+A*)/4          length
4     ATOM          name
1     BOOL          found
1               unused
2     CARD16          widthMM
2     CARD16          heightMM
2     p          nProperties
2     c          nColors
2     h          nShapes
2     s          nSections
2     d          nDoodads
2     a          nKeyAliases
1     CARD8          baseColorNdx
1     CARD8          labelColorNdx
f     KB_COUNTED_STRING16          labelFont
8p     LISTofKB_PROPERTY          properties
C0+..Cc     LISTofKB_COUNTED_STRING16          colors
H0+..Hh     LISTofKB_SHAPE          shapes
S0+..Ss     LISTofKB_SECTION          sections
D0+..Dd     LISTofKB_DOODAD          doodads
A0+..Aa     LISTofKB_KEYALIAS          keyAliases

KB_PROPERTY     4+n+v
2     n          nameLength
n     STRING8          name
2     v          valueLength
v     STRING8          value

KB_SHAPE     8+O*
```

```
4      ATOM          name
1      o          nOutlines
1      CARD8          primaryNdx
1      CARD8          approxNdx
1               unused
O0+..Oo     LISTofKB_OUTLINE          outlines

KB_OUTLINE     4+4p
1      p          nPoints
1      CARD8          cornerRadius
2               unused
4p     LISTofKB_POINT          points

KB_POINT
2      INT16          x
2      INT16          y

KB_SECTION     20+R*+D*+O*
4      ATOM          name
2      INT16          top
2      INT16          left
2      CARD16          width
2      CARD16          height
2      INT16          angle
1      CARD8          priority
1      r          nRows
1      d          nDoodads
1      o          nOverlays
2               unused
R0+..Rr     LISTofKB_ROW          rows
D0+..Dd     LISTofKB_DOODAD          doodads
O0+..Oo     LISTofKB_OVERLAY          overlays

KB_ROW     8+8k
2      INT16          top
2      INT16          left
1      k          nKeys
1      BOOL          vertical
2               unused
8k     LISTofKB_KEY          keys

KB_KEY
4      STRING8          name
2      INT16          gap
1      CARD8          shapeNdx
1      CARD8          colorNdx

KB_OVERLAY     8+R*
4      ATOM          name
1      r          nRows
3               unused
R0+..Rr     LISTofKB_OVERLAYROW          rows

KB_OVERLAYROW     4+8k
1      CARD8          rowUnder
```

```
1     k           nKeys
2               unused
8k     LISTofKB_OVERLAYKEY          keys

KB_OVERLAYKEY
4     STRING8           over
4     STRING8           under

KB_SHAPEDOODAD
4     ATOM          name
1     CARD8           type

KB_SHAPEDOODAD
     #1     XkbOutlineDoodad
     #2     XkbSolidDoodad
1     CARD8           priority
2     INT16           top
2     INT16           left
2     INT16           angle
1     CARD8           colorNdx
1     CARD8           shapeNdx
6               unused

KB_TEXTDOODAD     20+t+f
4     ATOM          name
1     CARD8           type
     #3     XkbTextDoodad
1     CARD8           priority
2     INT16           top
2     INT16           left
2     INT16           angle
2     CARD16           width
2     CARD16           height
1     CARD8           colorNdx
3               unused
t     KB_COUNTED_STRING16          text
f     KB_COUNTED_STRING16          font

KB_INDICATORDOODAD
4     ATOM          name
1     CARD8           type
     #4     XkbIndicatorDoodad
1     CARD8           priority
2     INT16           top
2     INT16           left
2     INT16           angle
1     CARD8           shapeNdx
1     CARD8           onColorNdx
1     CARD8           offColorNdx
5               unused

KB_LOGODOODAD     20+n
4     ATOM          name
1     CARD8           type
     #5     XkbLogoDoodad
```

```
1     CARD8          priority
2     INT16          top
2     INT16          left
2     INT16          angle
1     CARD8          colorNdx
1     CARD8          shapeNdx
6               unused
n     KB_COUNTED_STRING16          logoName

KB_DOODAD:
     KB_SHAPEDOODAD, or KB_TEXTDOODAD, or
     KB_INDICATORDOODAD, or KB_LOGODOODAD

1     CARD8          opcode
1     20          xkb-opcode
2     7+(f+8p+C*+H*+S*+D*+A*)/4          request-length
2     KB_DEVICESPEC          deviceSpec
1     h          nShapes
1     s          nSections
4     ATOM          name
2     CARD16          widthMM
2     CARD16          heightMM
2     p          nProperties
2     c          nColors
2     d          nDoodads
2     a          nKeyAliases
1     CARD8          baseColorNdx
1     CARD8          labelColorNdx
2               unused
f     KB_COUNTED_STRING16          labelFont
8p     LISTofKB_PROPERTY          properties
C0+..Cc     LISTofKB_COUNTED_STRING16          colors
H0+..Hh     LISTofKB_SHAPE          shapes
S0+..Ss     LISTofKB_SECTION          sections
D0+..Dd     LISTofKB_DOODAD          doodads
A0+..Aa     LISTofKB_KEYALIAS          keyAliases

1     CARD8          opcode
1     21          xkb-opcode
2     7          request-length
2     KB_DEVICESPEC          deviceSpec
2               unused
4     SETofKB_PERCLIENTFLAG          change
4     SETofKB_PERCLIENTFLAG          value
4     SETofKB_BOOLCTRL          ctrlsToChange
4     SETofKB_BOOLCTRL          autoCtrls
4     SETofKB_BOOLCTRL          autoCtrlValues

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     0          length
4     SETofKB_PERCLIENTFLAG          supported
4     SETofKB_PERCLIENTFLAG          value
4     SETofKB_BOOLCTRL          autoCtrls
```

```
4     SETofKB_BOOLCTRL          autoCtrlValues
8               unused

1     CARD8          opcode
1     22          xkb-opcode
2     2+(6+m+k+t+c+s+g+p)/4          request-length
2     KB_DEVICESPEC           deviceSpec
2     CARD16          maxNames
1     m          keymapsSpecLen
m     STRING          keymapsSpec
1     k          keycodesSpecLen
k     STRING          keycodesSpec
1     t          typesSpecLen
t     STRING          typesSpec
1     c          compatMapSpecLen
c     STRING          compatMapSpec
1     s          symbolsSpecLen
s     STRING          symbolsSpec
1     g          geometrySpecLen
g     STRING          geometrySpec
p               unused,p=pad(6+m+k+t+c+s+g)

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     (M*+K*+T*+C*+S*+G*+p)/4          length
2     m          nKeymaps
2     k          nKeycodes
2     t          nTypes
2     c          nCompatMaps
2     s          nSymbols
2     g          nGeometries
2     CARD16          extra
10               unused
M0+..Mm     LISTofKB_LISTING          keymaps
K0+..Kk     LISTofKB_LISTING          keycodes
T0+..Tt     LISTofKB_LISTING          types
C0+..Cc     LISTofKB_LISTING          compatMaps
S0+..Ss     LISTofKB_LISTING          symbols
G0+..Gg     LISTofKB_LISTING          geometries
p               unused,p=pad(M*+K*+T*+C*+S*+G*)

KB_LISTING     4+n+p
2     CARD16          flags
2     n          length
n     STRING8          string
p               unused,p=pad(n) to a 2-byte boundary

1     CARD8          opcode
1     23          xkb-opcode
2     3+(6+m+k+t+c+s+g+p)/4          request-length
2     KB_DEVICESPEC           deviceSpec
2     SETofKB_GBNDETAILMASK          need
2     SETofKB_GBNDETAILMASK          want
1     BOOL          load
```

```
1               unused
1     m          keymapsSpecLen
m     STRING8          keymapsSpec
1     k          keycodesSpecLen
k     STRING8          keycodesSpec
1     t          typesSpecLen
t     STRING8          typesSpec
1     c          compatMapSpecLen
c     STRING8          compatMapSpec
1     s          symbolsSpecLen
s     STRING8          symbolsSpec
1     g          geometrySpecLen
g     STRING8          geometrySpec
p               unused,p=pad(6+m+k+t+c+s+g)

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     V/4          length
1     KEYCODE          minKeyCode
1     KEYCODE          maxKeyCode
1     BOOL          loaded
1     BOOL          newKeyboard
2     SETofKB_GBNDETAILMASK          found
2     SETofKB_GBNDETAILMASK          reported
16               unused
V     LISTofITEMs          replies
     SETofKB_GBNDETAILMASK          (reported)
     XkbGBN_Types     map
     XkbGBN_CompatMap     compat
     XkbGBN_ClientSymbols     map
     XkbGBN_ServerSymbols     map
     XkbGBN_IndicatorMap     indicators
     XkbGBN_KeyNames     names
     XkbGBN_OtherNames     names
     XkbGBN_Geometry     geometry

ITEMs
M     XkbGetMap reply          map
C     XkbGetCompatMap reply          compat
I     XkbGetIndicatorMap reply          indicators
N     XkbGetNames reply          names
G     XkbGetGeometry reply          geometry

1     CARD8          opcode
1     24          xkb-opcode
2     4          request-length
2     KB_DEVICESPEC          deviceSpec
2     SETofKB_DEVFEATURE          wanted
1     BOOL          allButtons
1     CARD8          firstButton
1     CARD8          nButtons
1               unused
2     KB_LEDCLASSSPEC          ledClass
```

```
2     KB_IDSPEC          ledID

1     1          Reply
1     CARD8          deviceID
2     CARD16          sequence number
4     (2+n+p+8b+L*)/4          length
2     SETofKB_DEVFEATURE          present
2     SETofKB_FEATURE          supported
2     SETofKB_FEATURE          unsupported
2     l          nDeviceLedFBs
1     CARD8          firstBtnWanted
1     CARD8          nBtnsWanted
1     CARD8          firstBtnRtrn
1     b          nBtnsRtrn
1     CARD8          totalBtns
1     BOOL          hasOwnState
2     SETofKB_IDRESULT          dfltKbdFB
2     SETofKB_IDRESULT          dfltLedFB
2               unused
4     ATOM          devType
2     n          nameLen
n     STRING8          name
p               unused,p=pad(2+n)
8b     LISTofKB_ACTION          btnActions
L0+..Ll     LISTofKB_DEVICELEDINFO          leds

KB_DEVICELEDINFO     20+4n+12m
2     KB_LEDCLASSSPEC          ledClass
2     KB_IDSPEC          ledID
4     SETofKB_INDICATOR          namesPresent (has n bits set to 1)
4     SETofKB_INDICATOR          mapsPresent (has m bits set to 1)
4     SETofKB_INDICATOR          physIndicators
4     SETofKB_INDICATOR          state
4n     LISTofATOM          names
12m     LISTofKB_INDICATORMAP          maps

1     ??          opcode
1     25          xkb-opcode
2     3+(8b+L*)/4          request-length
2     KB_DEVICESPEC          deviceSpec
1     CARD8          firstBtn
1     b          nBtns
2     SETofKB_DEVFEATURE          change
2     l          nDeviceLedFBs
8b     LISTofKB_ACTION          btnActions
L0+..Ll     LISTofKB_DEVICELEDINFO          leds
     Encoding of KB_DEVICELEDINFO is as for XkbGetDeviceInfo

1     ??          opcode
1     101          xkb-opcode
2     6+(n+p)/4          request-length
2     n          msgLength
2               unused
4     CARD32          affectFlags
4     CARD32          flags
```

```
4     CARD32          affectCtrls
4     CARD32          ctrls
n     STRING8          message
p               unused, p=pad(n)

1     1          Reply
1               unused
2     CARD16          sequence number
4     0          length
4     CARD32          currentFlags
4     CARD32          currentCtrls
4     CARD32          supportedFlags
4     CARD32          supportedCtrls
8               unused
```

Events

```
1     ??          code
1     0          xkb code
2     CARD16          sequence number
4     TIMESTAMP          time
1     CARD8          deviceID
1     CARD8          oldDeviceID
1     KEYCODE          minKeyCode
1     KEYCODE          maxKeyCode
1     KEYCODE          oldMinKeyCode
1     KEYCODE          oldMaxKeyCode
1     CARD8          requestMajor
1     CARD8          requestMinor
2     SETofKB_NKNDETAIL          changed
14               unused

1     ??          code
1     1          xkb code
2     CARD16          sequence number
4     TIMESTAMP          time
1     CARD8          deviceID
1     SETofBUTMASK          ptrBtnActions
2     SETofKB_MAPPART          changed
1     KEYCODE          minKeyCode
1     KEYCODE          maxKeyCode
1     CARD8          firstType
1     CARD8          nTypes
1     KEYCODE          firstKeySym
1     CARD8          nKeySyms
1     KEYCODE          firstKeyAct
1     CARD8          nKeyActs
1     KEYCODE          firstKeyBehavior
1     CARD8          nKeyBehavior
1     KEYCODE          firstKeyExplicit
1     CARD8          nKeyExplicit
1     KEYCODE          firstModMapKey
1     CARD8          nModMapKeys
```

1	KEYCODE	firstVModMapKey
1	CARD8	nVModMapKeys
2	SETofKB_VMOD	virtualMods
2	unused	

1	??	code
1	2	xkb code
2	CARD16	sequence number
4	TIMESTAMP	time
1	CARD8	deviceID
1	SETofKEYMASK	mods
1	SETofKEYMASK	baseMods
1	SETofKEYMASK	latchedMods
1	SETofKEYMASK	lockedMods
1	KB_GROUP	group
2	INT16	baseGroup
2	INT16	latchedGroup
1	KB_GROUP	lockedGroup
1	SETofKEYMASK	compatState
1	SETofKEYMASK	grabMods
1	SETofKEYMASK	compatGrabMods
1	SETofKEYMASK	lookupMods
1	SETofKEYMASK	compatLookupMods
2	SETofBUTMASK	ptrBtnState
2	SETofKB_STATEPART	changed
1	KEYCODE	keycode
1	CARD8	eventType
1	CARD8	requestMajor
1	CARD8	requestMinor

1	??	code
1	3	xkb code
2	CARD16	sequence number
4	TIMESTAMP	time
1	CARD8	deviceID
1	CARD8	numGroups
2	unused	
4	SETofKB_CONTROL	changedControls
4	SETofKB_BOOLCTRL	enabledControls
4	SETofKB_BOOLCTRL	enabledControlChanges
1	KEYCODE	keycode
1	CARD8	eventType
1	CARD8	requestMajor
1	CARD8	requestMinor
4	unused	

1	??	code
1	4	xkb code
2	CARD16	sequence number
4	TIMESTAMP	time
1	CARD8	deviceID
3	unused	
4	SETofKB_INDICATOR	state
4	SETofKB_INDICATOR	stateChanged

12		unused

1	??	code
1	5	xkb code
2	CARD16	sequence number
4	TIMESTAMP	time
1	CARD8	deviceID
3		unused
4	SETofKB_INDICATOR	state
4	SETofKB_INDICATOR	mapChanged
12		unused

1	??	code
1	6	xkb code
2	CARD16	sequence number
4	TIMESTAMP	time
1	CARD8	deviceID
1		unused
2	SETofKB_NAMEDETAIL	changed
1	CARD8	firstType
1	CARD8	nTypes
1	CARD8	firstLevelName
1	CARD8	nLevelNames
1		unused
1	CARD8	nRadioGroups
1	CARD8	nKeyAliases
1	SETofKB_GROUP	changedGroupNames
2	SETofKB_VMOD	changedVirtualMods
1	KEYCODE	firstKey
1	CARD8	nKeys
4	SETofKB_INDICATOR	changedIndicators
4		unused

1	??	code
1	7	xkb code
2	CARD16	sequence number
4	TIMESTAMP	time
1	CARD8	deviceID
1	SETofKB_GROUP	changedGroups
2	CARD16	firstSI
2	CARD16	nSI
2	CARD16	nTotalSI
16		unused

1	??	code
1	8	xkb code
2	CARD16	sequence number
4	TIMESTAMP	time
1	CARD8	deviceID
1	KB_BELLCLASSRESULT	bellClass
1	CARD8	bellID
1	CARD8	percent
2	CARD16	pitch
2	CARD16	duration
4	ATOM	name

```
4     WINDOW          window
1     BOOL          eventOnly
7               unused

1     ??          code
1     9          xkb code
2     CARD16          sequence number
4     TIMESTAMP          time
1     CARD8          deviceID
1     KEYCODE          keycode
1     BOOL          press
1     BOOL          keyEventFollows
1     SETofKEYMASK          mods
1     KB_GROUP          group
8     STRING8          message
10               unused

1     ??          code
1     10          xkb code
2     CARD16          sequence number
4     TIMESTAMP          time
1     CARD8          deviceID
1     KEYCODE          keycode
2     SETofKB_AXNDETAIL          detail
2     CARD16          slowKeysDelay
2     CARD16          debounceDelay

1     ??          code
16               unused

1     ??          code
1     11          xkb code
2     CARD16          sequence number
4     TIMESTAMP          time
1     CARD8          deviceID
1               unused
2     SETofKB_XIDETAIL          reason
2     KB_LEDCLASSRESULT          ledClass
2     CARD8          ledID
4     SETofKB_INDICATOR          ledsDefined
4     SETofKB_INDICATOR          ledState
1     CARD8          firstButton
1     CARD8          nButtons
2     SETofKB_XIFEATURE          supported
2     SETofKB_XIFEATURE          unsupported
2               unused
```