

The X Keyboard Extension: Library Specification

Library Version 1.0 / Document Revision 1.0

X Consortium Standard

X Version 11, Release 6.1

Amber J. Benson and Gary Aitken

Erik Fortune
Silicon Graphics, Inc.

Donna Converse
X Consortium Inc.

George Sachs
Hewlett-Packard Company

Will Walker
Digital Equipment Corporation

Copyright © 1995, 1996 X Consortium Inc.

Copyright © 1995, 1996 Silicon Graphics Inc.

Copyright © 1995, 1996 Hewlett-Packard Company

Copyright © 1995, 1996 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.

Acknowledgments

This document is the result of a great deal of hard work by a great many people. Without Erik Fortune's work as Architect of the X Keyboard Extension and the longtime support of Silicon Graphics Inc. there would not be a keyboard extension.

We gratefully thank Will Walker and George Sachs for their help and expertise in providing some of the content for this document, and Digital Equipment Corporation and Hewlett-Packard for allowing them to participate in this project, and we are deeply indebted to IBM for providing the funding to complete this library specification.

Most of all, we thank Gary Aitken and Amber J. Benson for their long hours and late nights as ultimate authors of this specification, and for serving as authors, document editors, and XKB protocol and implementation reviewers. Their commitment to accuracy and completeness, their attention to detail, their keen insight, and their good natures when working under tremendous pressure are in some measure responsible not only for the quality of this document, but for the quality of the Keyboard extension itself.

Matt Landau
Manager, X Window System
X Consortium Inc.
5 February 1996

The following table shows the font conventions used in this document:

<i>Usage</i>	<i>Font example</i>
Key Labels	<code>Num_Lock</code>
New terms	<i>SlowKeys acceptance delay</i>
Function definitions	<code>XkbColorPtr XkbAddGeomColor(geom,spec,pixel)</code>
Function references	<code>XkbAddGeomColor</code>
Parameters or arguments	<i>geom</i>
Structure definitions	XkbGeometryRec
Structure references	<code>XkbGeometryRec</code>
References to fields in a data structure	<i>key_aliases</i>
References to masks, modifiers, controls	<code>IgnoreGroupLock</code>

1	Overview	1
1.1	Core X Protocol Support for Keyboards	1
1.2	Xkb Keyboard Extension Support for Keyboards.....	1
1.3	Xkb Extension Components.....	1
1.3.1	Groups and Shift Levels.....	3
1.3.2	Radio Groups	3
1.4	Client Types.....	3
1.5	Compatibility with the Core Protocol	4
1.6	Additional Protocol Errors	4
1.7	Extension Library Functions	4
1.7.1	Error Indications	4
2	Initialization and General Programming Information.....	6
2.1	Extension Header Files.....	6
2.2	Extension Name	6
2.3	Determining Library Compatibility	6
2.4	Initializing the Keyboard Extension.....	7
2.5	Disabling the Keyboard Extension.....	8
2.6	Protocol Errors	9
2.7	Display and Device Specifications in Function Calls	9
3	Data Structures	11
3.1	Allocating Xkb Data Structures	11
3.2	Adding Data and Editing Data Structures	11
3.3	Making Changes to the Server's Keyboard Description	12
3.4	Tracking Keyboard Changes in the Server.....	12
3.5	Freeing Data Structures	13
4	Xkb Events.....	14
4.1	Xkb Event Types	14
4.2	Xkb Event Data Structures	15
4.3	Selecting Xkb Events	15
4.3.1	Event Masks.....	17
4.4	Unified Xkb Event Type.....	18
5	Keyboard State	19
5.1	Keyboard State Description	19
5.2	Changing the Keyboard State.....	22
5.2.1	Changing Modifiers	22
5.2.2	Changing Groups	23
5.3	Determining Keyboard State	23
5.4	Tracking Keyboard State.....	24
6	Complete Keyboard Description.....	27
6.1	The XkbDescRec Structure	27
6.2	Obtaining a Keyboard Description from the Server.....	28
6.3	Tracking Changes to the Keyboard Description in the Server	28
6.4	Allocating and Freeing a Keyboard Description.....	28
7	Virtual Modifiers	30
7.1	Virtual Modifier Names and Masks	30

7.2	Modifier Definitions	30
7.3	Binding Virtual Modifiers to Real Modifiers	31
7.4	Virtual Modifier Key Mapping.....	31
7.4.1	Inactive Modifier Sets	32
7.5	Conventions.....	32
7.6	Example.....	32
8	Indicators.....	34
8.1	Indicator Names	34
8.2	Indicator Data Structures.....	34
8.2.1	XkbIndicatorRec.....	34
8.2.2	XkbIndicatorMapRec.....	35
8.3	Getting Information About Indicators.....	39
8.3.1	Getting Indicator State	40
8.3.2	Getting Indicator Information by Index.....	40
8.3.3	Getting Indicator Information by Name	40
8.4	Changing Indicator Maps and State	41
8.4.1	Effects of Explicit Changes on Indicators	41
8.4.2	Changing Indicator Maps by Index.....	42
8.4.3	Changing Indicator Maps by Name	43
8.4.4	The XkbIndicatorChangesRec Structure	43
8.5	Tracking Changes To Indicator State or Map.....	44
8.6	Allocating and Freeing Indicator Maps.....	45
9	Bells	47
9.1	Bell Names	47
9.2	Audible Bells.....	48
9.3	Bell Functions	48
9.3.1	Generating Named Bells	49
9.3.2	Generating Named Bell Events.....	50
9.3.3	Forcing a Server-generated Bell	51
9.4	Detecting Bells	51
10	Keyboard Controls	53
10.1	Controls that Enable and Disable other Controls	54
10.1.1	The EnabledControls Control	54
10.1.2	The AutoReset Control	54
10.2	Control for Bell Behavior.....	56
10.2.1	The AudibleBell Control.....	56
10.3	Controls for Repeat Key Behavior	56
10.3.1	The PerKeyRepeat Control	56
10.3.2	The RepeatKeys Control.....	56
10.3.3	The DetectableAutorepeat Control	57
10.4	Controls for Keyboard Overlays (Overlay1 and Overlay2 Controls)	58
10.5	Controls for Using the Mouse from the Keyboard.....	59
10.5.1	The MouseKeys Control	59
10.5.2	The MouseKeysAccel Control.....	59
10.6	Controls for Better Keyboard Access by Physically-Impaired Persons.....	61
10.6.1	The AccessXKeys Control.....	61
10.6.2	The AccessXTIMEout Control	62
10.6.3	The AccessXFeedback Control.....	63
10.6.4	AccessXNotify Events	64

10.6.5	StickyKeys, RepeatKeys, and MouseKeys Events	65
10.6.6	The SlowKeys Control.....	65
10.6.7	The BounceKeys Control.....	66
10.6.8	The StickyKeys Control.....	67
10.7	Controls for General Keyboard Mapping.....	68
10.7.1	The GroupsWrap Control.....	69
10.7.2	The IgnoreLockMods Control	69
10.7.3	The IgnoreGroupLock Control	70
10.7.4	The InternalMods Control.....	70
10.8	The XkbControlsRec Structure	71
10.9	Querying Controls	76
10.10	Changing Controls.....	77
10.10.1	The XkbControlsChangesRec Structure	78
10.11	Tracking Changes to Keyboard Controls	78
10.12	Allocating and Freeing an XkbControlsRec	80
11	X Library Controls.....	81
11.1	Controls Affecting Keycode-to-String Translation	81
11.1.1	ForceLatin1Lookup.....	81
11.1.2	ConsumeLookupMods	81
11.1.3	AlwaysConsumeShiftAndLock	82
11.2	Controls Affecting Compose Processing	82
11.2.1	ConsumeKeysOnComposeFail	82
11.2.2	ComposeLED.....	83
11.2.3	BeepOnComposeFail	83
11.3	Controls Effecting Event Delivery	83
11.3.1	IgnoreNewKeyboards	83
11.4	Manipulating the Library Controls.....	84
11.4.1	Determining which Library Controls are Implemented.....	84
11.4.2	Determining the State of the Library Controls	84
11.4.3	Changing the State of the Library Controls	84
12	Interpreting Key Events	86
12.1	Effects of Xkb on the Core X Library	86
12.1.1	Effects of Xkb on Event State	86
12.1.2	Effects of Xkb on MappingNotify Events	86
12.1.3	X Library Functions Affected by Xkb	87
12.2	Xkb Event and Keymap Functions.....	88
13	Keyboard Geometry.....	91
13.1	Shapes and Outlines	93
13.2	Sections	94
13.3	Rows and Keys.....	94
13.4	Doodads.....	95
13.5	Overlay Rows and Overlay Keys	95
13.6	Drawing a Keyboard Representation	96
13.7	Geometry Data Structures	97
13.8	Getting Keyboard Geometry From the Server	103
13.9	Using Keyboard Geometry.....	104
13.10	Adding Elements to a Keyboard Geometry	105
13.11	Allocating and Freeing Geometry Components	109

14	Xkb Keyboard Mapping	115
14.1	Notation and Terminology.....	115
14.1.1	Core Implementation	116
14.1.2	Xkb Implementation	116
14.2	Getting Map Components from the Server	117
14.3	Changing Map Components in the Server	119
14.3.1	The XkbMapChangesRec Structure	119
14.4	Tracking Changes to Map Components	121
14.5	Allocating and Freeing Client and Server Maps	122
14.5.1	Allocating an Empty Client Map	122
14.5.2	Freeing a Client Map	123
14.5.3	Allocating an Empty Server Map	123
14.5.4	Freeing a Server Map.....	124
15	Xkb Client Keyboard Mapping.....	125
15.1	The XkbClientMapRec Structure.....	126
15.2	Key Types.....	126
15.2.1	The Canonical Key Types	128
15.2.2	Getting Key Types from the Server	130
15.2.3	Changing the Number of Levels in a Key Type	131
15.2.4	Copying Key Types.....	131
15.3	Key Symbol Map	132
15.3.1	Per-Key Key Type Indices	133
15.3.2	Per-Key Group Information.....	133
15.3.3	Key Width	134
15.3.4	Offset in to the Symbol Map.....	134
15.3.5	Getting the Symbol Map for Keys from the Server.....	136
15.3.6	Changing the Number of Groups and Types Bound to a Key	136
15.3.7	Changing the Number of Symbols Bound to a Key	137
15.4	The Per-Key Modifier Map	138
15.4.1	Getting the Per-Key Modifier Map from the Server	138
16	Xkb Server Keyboard Mapping	139
16.1	Key Actions.....	140
16.1.1	The XkbAction Structure.....	141
16.1.2	The XkbAnyAction Structure	142
16.1.3	Actions for Changing Modifiers' State	142
16.1.4	Actions for Changing Group State.....	144
16.1.5	Actions for Moving the Pointer	146
16.1.6	Actions for Simulating Pointer Button Press and Release.....	147
16.1.7	Actions for Changing the Pointer Button Simulated	148
16.1.8	Actions for Locking Modifiers and Group.....	149
16.1.9	Actions for Changing the Active Screen	152
16.1.10	Actions for Changing Boolean Controls State.....	153
16.1.11	Actions for Generating Messages	154
16.1.12	Actions for Generating a Different Keycode	155
16.1.13	Actions for Generating DeviceButtonPress and DeviceButtonRelease.....	157
16.1.14	Actions for Simulating Events from Device Valuators.....	158
16.1.15	Obtaining Key Actions for Keys from the Server	159
16.1.16	Changing the Number of Actions Bound to a Key	159
16.2	Key Behavior.....	160
16.2.1	Radio Groups	160
16.2.2	The XkbBehavior Structure	160

16.3	16.2.3 Obtaining Key Behaviors for Keys from the Server.....	161
16.3	Explicit Components—Avoiding Automatic Remapping by the Server.....	162
16.3.1	Obtaining Explicit Components for Keys from the Server.....	162
16.4	Virtual Modifier Mapping	163
16.4.1	Obtaining Virtual Modifier Bindings from the Server.....	164
16.4.2	Obtaining Per-Key Virtual Modifier Mappings from the Server	165
17	17. The Xkb Compatibility Map.....	166
17.1	17.1 The XkbCompatMap Structure	168
17.1.1	17.1.1 Xkb State to Core Protocol State Transformation	168
17.1.2	17.1.2 Core Keyboard Mapping to Xkb Keyboard Mapping Transformation.....	169
17.1.3	17.1.3 Xkb Keyboard Mapping to Core Keyboard Mapping Transformations	172
17.2	17.2 Getting Compatibility Map Components From the Server	173
17.3	17.3 Using the Compatibility Map	174
17.4	17.4 Changing the Server’s Compatibility Map.....	176
17.5	17.5 Tracking Changes to the Compatibility Map	177
17.6	17.6 Allocating and Freeing Compatibility Map	178
18	18. Symbolic Names	179
18.1	18.1 The XkbNamesRec Structure.....	179
18.2	18.2 Symbolic Names Masks	181
18.3	18.3 Getting Symbolic Names From the Server	182
18.4	18.4 Changing Symbolic Names on the Server.....	182
18.5	18.5 Tracking Name Changes	184
18.6	18.6 Allocating and Freeing Symbolic Names.....	185
19	19. Replacing a Keyboard “On the Fly”	186
20	20. Server Database of Keyboard Components	189
20.1	20.1 Component Names	190
20.2	20.2 Listing the Known Keyboard Components	190
20.3	20.3 Component Hints.....	191
20.4	20.4 Building a Keyboard Description using the Server Database	192
21	21. Attaching Xkb Actions to X Input Extension Devices	197
21.1	21.1 XkbDeviceInfoRec	198
21.2	21.2 Querying Xkb Features for Non-KeyClass Input Extension Devices.....	199
21.3	21.3 Allocating, Initializing, and Freeing the XkbDeviceInfoRec Structure.....	202
21.4	21.4 Setting Xkb Features for Non-KeyClass Input Extension Devices.....	203
21.5	21.5 XkbExtensionDeviceNotify Event	205
21.6	21.6 Tracking Changes to Extension Devices	206
22	22. Debugging Aids	209
	Glossary	211

Figure 1.1	Overall Xkb Structure	2
Figure 5.1	Xkb State.....	19
Figure 10.1	MouseKeys Acceleration.....	61
Figure 13.1	Rotated Keyboard Sections.....	91
Figure 13.2	Keyboard with Four Sections.....	93
Figure 13.3	Rows in a Section.....	94
Figure 13.4	Xkb Geometry Data Structures.....	97
Figure 13.5	Xkb Geometry Data Structures (Doodads)	98
Figure 13.6	Xkb Geometry Data Structures (Overlays).....	99
Figure 13.7	Key Surface, Shape Outlines and Bounding Box	104
Figure 14.1	Shift Levels and Groups.....	116
Figure 15.1	Xkb Client Map.....	125
Figure 16.1	Server Map Relationships	139
Figure 16.2	Virtual Modifier Relationships.....	164
Figure 17.1	Server Interaction with Types of Clients	166
Figure 17.2	Server Derivation of State and Keyboard Mapping Components	167
Figure 17.3	Xkb Compatibility Data Structures.....	168
Figure 20.1	Building a new keyboard description from the server database	195

Table 1.1	Function Error Returns due to Extension Problems	5
Table 2.1	Xkb Protocol Errors	9
Table 2.2	BadKeyboard Protocol Error resource_id Values	9
Table 4.1	Xkb Event Types	14
Table 4.2	XkbSelectEvents Mask Constants	17
Table 5.1	Real Modifier Masks	22
Table 5.2	Symbolic Group Names	23
Table 5.3	XkbStateNotify Event Detail Masks	24
Table 6.1	XkbDescRec Component References	27
Table 6.2	Mask Bits for XkbDescRec	28
Table 8.1	XkbIndicatorMapRec flags Field	35
Table 8.2	XkbIndicatorMapRec which_groups and groups, Keyboard Drives Indicator ..	37
Table 8.3	XkbIndicatorMapRec which_groups and groups, Indicator Drives Keyboard ..	37
Table 8.4	XkbIndicatorMapRec which_mods and mods, Keyboard Drives Indicator ..	38
Table 8.5	XkbIndicatorMapRec which_mods and mods, Indicator Drives Keyboard ..	39
Table 9.1	Predefined Bells	48
Table 9.2	Bell Sounding and Bell Event Generating	49
Table 10.1	Xkb Keyboard Controls	53
Table 10.2	MouseKeysAccel Fields	59
Table 10.3	AccessXFeedback Masks	63
Table 10.4	AccessXNotify Events	64
Table 10.5	AccessXNotify Event Details	65
Table 10.6	Xkb Controls]	72
Table 10.7	Controls Mask Bits	73
Table 10.8	GroupsWrap options (groups_wrap field)	74
Table 10.9	Access X Enable/Disable Bits (ax_options field)	75
Table 11.1	Library Control Masks	84
Table 13.1	Doodad Types	95
Table 14.1	Xkb Mapping Component Masks and Convenience Functions	118
Table 14.2	XkbMapChangesRec Masks	120
Table 14.3	XkbAllocClientMap Masks	122
Table 14.4	XkbAllocServerMap Masks	124
Table 15.1	Example Key Type	127
Table 15.2	group_info Range Normalization	134
Table 15.3	Group Index Constants	137
Table 16.1	Action Types	142
Table 16.2	Modifier Action Types	143
Table 16.3	Modifier Action Flags	144
Table 16.4	Group Action Types	145
Table 16.5	Group Action Flags	145
Table 16.6	Pointer Action Types	146

Table 16.7	Pointer Button Action Types.....	148
Table 16.8	Pointer Button Action Flags	148
Table 16.9	Pointer Default Flags	149
Table 16.10	ISO Action Flags when XkbSA_ISODfltIsGroup is Set	150
Table 16.11	ISO Action Flags when XkbSA_ISODfltIsGroup is Not Set	151
Table 16.12	ISO Action affect Field Values	151
Table 16.13	Switch Screen Action Flags	152
Table 16.14	Controls Action Types	153
Table 16.15	Control Action Flags.....	153
Table 16.16	Message Action Flags	154
Table 16.17	Device Button Action Types	157
Table 16.18	Device Button Action Flags.....	157
Table 16.19	Device Valuator v<n>_what High Bits Values	158
Table 16.20	Key Behaviors.....	160
Table 16.21	Explicit Component Masks.....	162
Table 17.1	Symbol Interpretation Match Criteria.....	171
Table 17.2	Compatibility Map Component Masks.....	173
Table 18.1	Symbolic Names Masks.....	181
Table 18.2	XkbNameChanges Fields.....	183
Table 19.1	XkbNewKeyboardNotifyEvent Details	187
Table 20.1	Server Database Keyboard Components	189
Table 20.2	XkbComponentNameRec flags bits	192
Table 20.3	want and need mask bits and required names components.....	194
Table 20.4	XkbDescRec components returned for values of want & need s.....	196
Table 21.1	XkbDeviceInfoRec Mask Bits	199
Table 22.1	Debug Control Masks	209

1 Overview

The X Keyboard Extension provides capabilities that are lacking or are cumbersome in the core X protocol.

1.1 Core X Protocol Support for 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.
- A second keyboard group may be specified using a modifier, but this has side-effects that wreak havoc with client grabs and X toolkit translations. Furthermore, this approach limits the number of keyboard groups to two.
- 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 the shift modifier should “cancel” the lock modifier.
- The lack of any explicit descriptions for indicators, most modifiers and other aspects of the keyboard appearance requires clients that wish to clearly describe the keyboard to a user to resort to a mish-mash of prior knowledge and heuristics.

1.2 Xkb Keyboard Extension Support for Keyboards

The X Keyboard 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 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, Xkb provides descriptions and symbolic names for many aspects of keyboard appearance and behavior.

In addition, the X Keyboard Extension includes additional keyboard controls designed to make keyboards more accessible to people with movement impairments.

1.3 Xkb Extension Components

The Xkb extension is composed of two parts: a server extension, and a client-side X library extension. These consist of a loadable module which may be activated when an X server is started, and a modified version of Xlib. Both server and Xlib versions must be at least X11 R6.

The diagram below shows the overall structure of the Xkb extension:

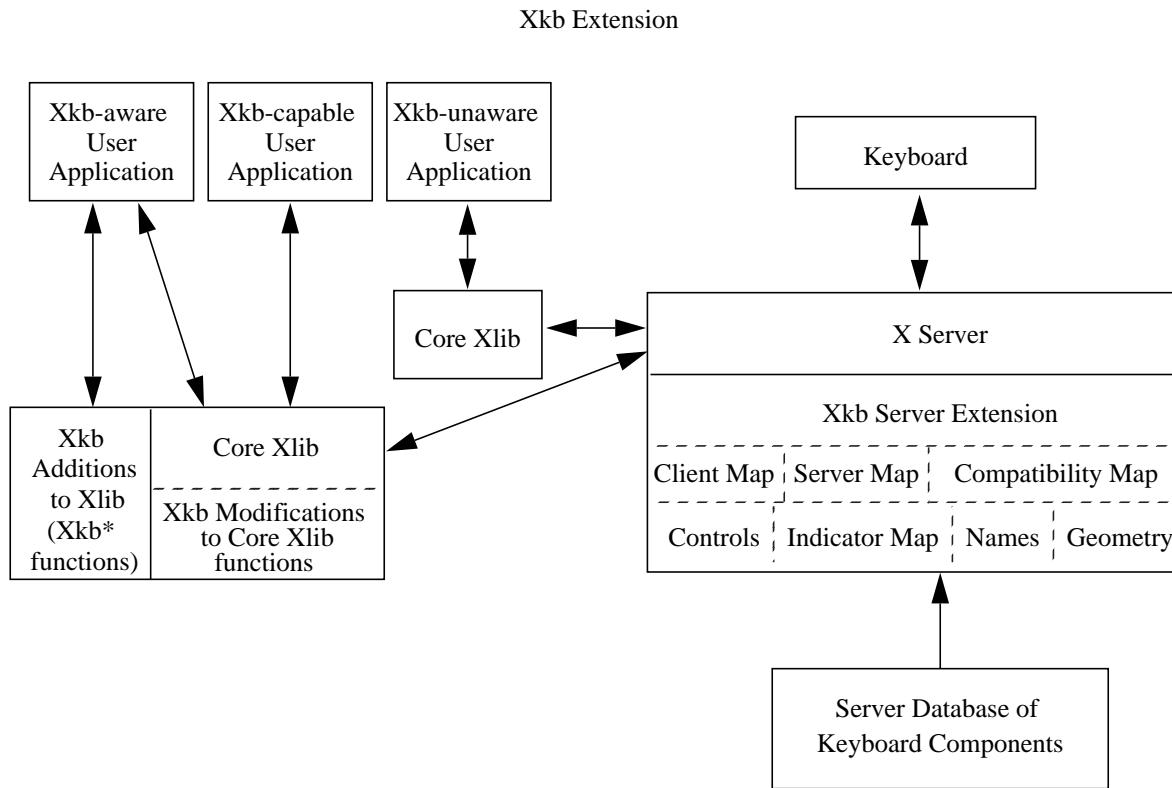


Figure 1.1 Overall Xkb Structure

The server portion of the Xkb extension encompasses a database of named keyboard components, in unspecified format, which may be used to configure a keyboard. Internally, the server maintains a *keyboard description* which includes the keyboard state and configuration (mapping). By “keyboard” we mean the logical keyboard device, which includes not only the physical keys, but also potentially a set of up to 32 indicators (usually LEDs) and bells.

The keyboard description is a composite of several different data structures, each of which may be manipulated separately. When manipulating the server components, the design allows partial components to be transmitted between the server and a client. The individual components are shown in Figure 1.1 above; they are:

Client Map

The key mapping information needed to convert arbitrary keycodes to symbols.

Server Map

The key mapping information categorizing keys by functionality (which keys are modifiers, how keys behave, etc.).

Controls

Client configurable quantities effecting how the keyboard behaves, such as repeat behavior and modifications for people with movement impairments.

Indicators

The mapping of behavior to indicators.

Geometry

A complete description of the physical keyboard layout, sufficient to draw a representation of the keyboard.

Names

A mapping of names to various aspects of the keyboard such as individual virtual modifiers, indicators, and bells.

Compatibility Map

The definition of how to map core protocol keyboard state to Xkb keyboard state.

A client application interrogates and manipulates the keyboard by reading and writing portions of the server description for the keyboard. In a typical sequence a client would fetch the current information it is interested in, modify it, and write it back. If a client wishes to track some portion of the keyboard state, it typically maintains a local copy of the portion of the server keyboard description dealing with the items of interest, and updates this local copy from events describing state transitions which are sent by the server.

A client may request the server to reconfigure the keyboard either by sending explicit reconfiguration instructions to it, or by telling it to load a new configuration from its database of named components. Partial reconfiguration and incremental reconfiguration are both supported.

1.3.1 Groups and Shift Levels

The graphic characters or control functions that may be accessed by one key are logically arranged in groups and levels. See section 14.1 on page 116 for a complete description of groups and levels.

1.3.2 Radio Groups

A radio group is a set of keys whose behavior simulates a set of radio buttons. Once a key in a radio group is pressed, it stays logically depressed until another key in the group is pressed, at which point the previously depressed key is logically released. Consequently, at most one key in a radio group can be logically depressed at one time. A radio group is defined by a radio group index, an optional name, and by assigning each key in the radio group `XkbKB_RadioGroup` behavior and the radio group index.

1.4 Client Types

This specification differentiates between three different classes of client applications:

- Xkb-aware applications
These applications make specific use of Xkb functionality and APIs not present in the core protocol.
- Xkb-capable applications

These applications make no use of Xkb extended functionality and APIs directly. However, they are linked with a version of Xlib which includes Xkb, and indirectly benefit from some of Xkb's features.

- Xkb-unaware applications

These applications make no use of Xkb extended functionality or APIs, and require Xkb's functionality to be mapped to core Xlib functionality to operate properly.

1.5 Compatibility with the Core Protocol

Because the Xkb extension allows a keyboard to be configured in ways not foreseen by the core protocol, and because Xkb-unaware clients are allowed to connect to a server using the Xkb extension, there must be a means of converting between the Xkb domain and the core protocol. The Xkb server extension maintains a compatibility map as part of its keyboard description; this map controls conversion of Xkb generated events to core protocol events, and the results of core protocol requests to appropriate Xkb state and configuration.

1.6 Additional Protocol Errors

The Xkb extension adds a single protocol error, `BadKeyboard`, to the core protocol error set. See section 2.6 for a discussion of the `BadKeyboard` protocol error.

1.7 Extension Library Functions

The X Keyboard Extension replaces the core protocol definition of a keyboard with a more comprehensive one. The X Keyboard Extension library interfaces are included in Xlib.¹

Xlib detects the presence of the X Keyboard server extension, and uses Xkb protocol to replace some standard X library functions related to the keyboard. If an application uses only standard X library functions to examine the keyboard or process key events, it should not need to be modified when linked with an X library containing the X keyboard extension. All of the keyboard-related X library functions have been modified to automatically use Xkb protocol when the server extension is present.

The Xkb extension adds library interfaces to allow a client application to directly manipulate the new capabilities.

1.7.1 Error Indications

Xkb functions which communicate with the X server check to be sure the Xkb extension has been properly initialized prior to doing any other operations. If the extension has not been properly initialized or the application, library and server versions are incompatible, these functions return an error indication as shown in Table 1.1. Because of this test,

1. X11R6.1 is the first release by the X Consortium, Inc. which includes the X Keyboard Extension in Xlib. X11R6 included work in progress on this extension as non-standard additions to the library.

BadAccess and BadMatch (due to incompatible versions) protocol errors should normally not be generated.

Table 1.1 Function Error Returns due to Extension Problems

<i>Functions return type</i>	<i>Return value</i>
pointer to a structure	NULL
Bool	False
Status	BadAccess

Many Xkb functions do not actually communicate with the X server; they only require processing in the client-side portion of the library. Furthermore, some applications may never actually need to communicate with the server; they simply use the Xkb library capabilities. The functions which do not communicate with the server return either a pointer to a structure, a Bool, or a Status. These functions check that the application has queried the Xkb library version and return the values shown in Table 1.1 if it has not.

2 Initialization and General Programming Information

2.1 Extension Header Files

The following include files are part of the Xkb standard:

- <X11/XKbLib.h>
`XKbLib.h` is the main header file for Xkb; it declares constants, types, and functions.
- <X11/extensions/XKBstr.h>
`XKBstr.h` declares types and constants for Xkb. It is included automatically from <X11/XKbLib.h>; you should never need to reference it directly in your application code.
- <X11/extensions/XKB.h>
`XKB.h` defines constants for Xkb. It is included automatically from <X11/XKBstr.h>; you should never need to reference it directly in your application code.
- <X11/extensions/XKBgeom.h>
`XKBgeom.h` declares types, symbolic constants, and functions for manipulating keyboard geometry descriptions.

2.2 Extension Name

The name of the Xkb extension is given in <X11/extensions/Xkb.h>:

```
#define XkbName "XKEYBOARD"
```

Most extensions to the X protocol are initialized by calling *XInitExtension* and passing the extension name. However, as explained in section 2.4, Xkb requires a more complex initialization sequence, and a client program should not call *XInitExtension* directly.

2.3 Determining Library Compatibility

If an application is dynamically linked, both the X server and the client side X library must contain the Xkb extension in order for the client to use the Xkb extension capabilities. Therefore a dynamically linked application must check both the library and the server for compatibility before using Xkb function calls. A properly written program must check for compatibility between the version of the Xkb library which is dynamically loaded and the one used when the application was built. It must then check the server version for compatibility with the version of Xkb in the library.

If your application is statically linked, you must still check for server compatibility, and may check library compatibility. (It is possible to compile against one set of header files and link against a different, incompatible, version of the library, although this should not normally occur.)

To determine the compatibility of a library at runtime, call *XkbLibraryVersion*.

```
Bool XkbLibraryVersion(lib_major_in_out, lib_minor_in_out)
    int * lib_major_in_out; /* specifies and returns the major Xkb library version. */
    int * lib_minor_in_out; /* specifies and returns the minor Xkb library version. */
```

Pass the symbolic value `XkbMajorVersion` in `lib_major_in_out` and `XkbMinorVersion` in `lib_minor_in_out`. These arguments represent the version of the library used at compile time. The *XkbLibraryVersion* function backfills the major and minor version numbers of the library used at run time in `lib_major_in_out` and `lib_minor_in_out`. If the

versions of the compile time and run time libraries are compatible, *XkbLibraryVersion* returns True, otherwise it returns False.

In addition, in order to use the Xkb extension, you must insure the extension is present in the server and that the server supports the version of the extension expected by the client. Use *XkbQueryExtension* to do this, as described in the next section.

2.4 Initializing the Keyboard Extension

Call *XkbQueryExtension* to check for the presence and compatibility of the extension in the server and to initialize the extension. Because of potential version mismatches, you cannot use the generic extension mechanism functions (*XQueryExtension* and *XInitExtension*) for checking for the presence of, and initializing the Xkb extension.

You must call *XkbQueryExtension* or *XkbOpenDisplay* before using any other Xkb library interfaces, unless such usage is explicitly allowed in the interface description in this document. The exceptions are: *XkbIgnoreExtension*, *XkbLibraryVersion*, and a handful of audible-bell functions. You should not use any other Xkb functions if the extension is not present or is uninitialized. In general, calls to Xkb library routines made prior to initializing the Xkb extension cause BadAccess protocol errors.

XkbQueryExtension both determines whether a compatible Xkb extension is present in the X server and initializes the extension when it is present.

```
Bool XkbQueryExtension(dpy, opcode_rtrn, event_rtrn, error_rtrn, major_in_out,
                      minor_in_out)
    Display * dpy;          /* connection to the X server */
    int *     opcode_rtrn;   /* backfilled with the major extension opcode */
    int *     event_rtrn;    /* backfilled with the extension base event code */
    int *     error_rtrn;   /* backfilled with the extension base error code */
    int *     major_in_out;  /* compile time lib major version in, server major version out */
                           /* */
    int *     minor_in_out;  /* compile time lib min version in, server minor version out */
```

The *XkbQueryExtension* function determines whether a compatible version of the X Keyboard Extension is present in the server. If a compatible extension is present, *XkbQueryExtension* returns True; otherwise it returns False.

If a compatible version of Xkb is present, *XkbQueryExtension* initializes the extension. It backfills the major opcode for the keyboard extension in *opcode_rtrn*, the base event code in *event_rtrn*, the base error code in *error_rtrn*, and the major and minor version numbers of the extension in *major_in_out* and *minor_in_out*. The major opcode is reported in the *req_major* fields of some Xkb events. For a discussion of the base event code, see section 4.1 on page 15.

As a convenience, you can use the function *XkbOpenDisplay* to perform these three tasks at once: open a connection to an X server, check for a compatible version of the Xkb extension in both the library and the server, and initialize the extension for use.

```
Display *XkbOpenDisplay(display_name, event_rtrn, error_rtrn, major_in_out, minor_in_out,
                        reason_rtrn)
    char *display_name; /* hardware display name, which determines the display and
                           communications domain to be used */
    int * event_rtrn; /* backfilled with the extension base event code */
    int * error_rtrn; /* backfilled with the extension base error code */
    int * major_in_out; /* compile time lib major version in, server major version out */
    int * minor_in_out; /* compile time lib minor version in, server minor version out */
    int * reason_rtrn; /* backfilled with a status code */
```

XkbOpenDisplay is a convenience function which opens an X display connection and initializes the X keyboard extension. In all cases, upon return *reason_rtrn* contains a status value indicating success or the type of failure. If *major_in_out* and *minor_in_out* are not NULL, *XkbOpenDisplay* first calls *XkbLibraryVersion* to determine if the client library is compatible, passing it the values pointed to by *major_in_out* and *minor_in_out*. If the library is incompatible, *XkbOpenDisplay* backfills *major_in_out* and *minor_in_out* with the major and minor extension versions of the library being used and returns NULL. If the library is compatible, *XkbOpenDisplay* next calls *XOpenDisplay* with the *display_name*. If this fails, the function returns NULL. If successful, *XkbOpenDisplay* calls *XkbQueryExtension* and backfills the major and minor Xkb server extension version numbers in *major_in_out* and *minor_in_out*. If the server extension version is not compatible with the library extension version or if the server extension is not present, *XkbOpenDisplay* closes the display and returns NULL. When successful, the function returns the display connection.

The possible values for *reason_rtrn* are:

- *XkbOD_BadLibraryVersion* indicates *XkbLibraryVersion* returned False.
- *XkbOD_ConnectionRefused* indicates the display could not be opened.
- *XkbOD_BadServerVersion* indicates the library and the server have incompatible extension versions.
- *XkbOD_NonXkbServer* indicates the extension is not present in the X server.
- *XkbOD_Success* indicates that the function succeeded.

2.5 Disabling the Keyboard Extension

If a server supports the Xkb extension, the X library normally implements pre-Xkb keyboard functions using the Xkb keyboard description and state. The server Xkb keyboard state may differ from the pre-Xkb keyboard state. This difference does not affect most clients, but there are exceptions. To allow these clients to work properly, you may instruct the extension not to use Xkb functionality.

Call *XkbIgnoreExtension* to prevent core X library keyboard functions from using the X Keyboard Extension. You must call *XkbIgnoreExtension* before you open a server connection; Xkb does not provide a way to enable or disable use of the extension once a connection is established.

```
Bool XkbIgnoreExtension(ignore)
    Bool ignore; /* True means ignore the extension */
```

XkbIgnoreExtension tells the X library whether to use the X Keyboard Extension on any subsequently opened X display connections. If ignore is True, the library does not initialize the Xkb extension when it opens a new display. This forces the X server to use compatibility mode and communicate with the client using only core protocol requests and events. If ignore is False, the library treats subsequent calls to *XOpenDisplay* normally and uses Xkb extension requests, events and state. Do not explicitly use Xkb on a connection for which it is disabled. *XkbIgnoreExtension* returns False if it was unable to apply the ignore request.

2.6 Protocol Errors

Many of the Xkb extension library functions described in this document can cause the X server to report an error, referred to in this document as a *BadXxx* protocol error, where Xxx is some name. These errors are fielded in the normal manner, by the default Xlib error handler, or one replacing it. Note that X protocol errors are not necessarily reported immediately, due to the buffering of X protocol requests in Xlib and the server.

Table 2.1 lists the protocol errors that can be generated, and their causes.

Table 2.1 Xkb Protocol Errors

Error	Cause
BadAccess	The Xkb extension has not been properly initialized
BadKeyboard	The device specified was not a valid core or input extension device
BadImplementation	Invalid reply from server
BadAlloc	Unable to allocate storage
BadMatch	A compatible version of Xkb was not available in the server or an argument has correct type and range, but is otherwise invalid
BadValue	An argument is out of range
BadAtom	A name is neither a valid Atom or None
BadDevice	Device, Feedback Class, or Feedback Id invalid

The Xkb extension adds a single protocol error, *BadKeyboard*, to the core protocol error set. This error code will be reported as the *error_rtn* when *XkbQueryExtension* is called. When a *BadKeyboard* error is reported in an *XErrorEvent*, additional information is reported in the *resource_id* field. The most significant byte of the *resource_id* is a further refinement of the error cause, as defined in Table 2.2. The least significant byte will contain the device, class, or feedback id as indicated in the table.

Table 2.2 BadKeyboard Protocol Error resource_id Values

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

2.7 Display and Device Specifications in Function Calls

Where a connection to the server is passed as an argument (*Display**) and an *XkbDescPtr* is also passed as an argument, the *Display** argument must match the *dpy* field of the *XkbDescRec* pointed to by the *XkbDescPtr* argument, or else the *dpy* field of the *XkbDescRec* must be NULL. If they don't match or the *dpy* field is not NULL, a

BadMatch error is returned (either in the return value or a backfilled Status variable). Upon successful return, the *dpy* field of the XkbDescRec always contains the Display* value passed in.

The Xkb extension can communicate with the X input extension if it is present. Consequently, there can potentially be more than one input device connected to the server. Most Xkb library calls which require communicating with the server involve both a server connection (Display * *dpy*) and a device identifier (unsigned int *device_spec*). In some cases, the device identifier is implicit, and is taken as the *device_spec* field of an XkbDescRec structure passed as an argument.

The device identifier can specify any X input extension device with a KeyClass component, or it can specify the constant, XkbUseCoreKbd. The use of XkbUseCoreKbd allows applications to indicate the core keyboard without having to determine its device identifier.

Where an Xkb device identifier is passed as an argument and an XkbDescPtr is also passed as an argument, if either the argument or the XkbDescRec *device_spec* field is XkbUseCoreKbd, and if the function returns successfully, the XkbDescPtr *device_spec* field will have been converted from XkbUseCoreKbd to a real Xkb device id. If the function does not complete successfully, the *device_spec* field remains unchanged. Subsequently, the device id argument must match the *device_spec* field of the XkbDescPtr argument. If they don't match, a BadMatch error is returned (either in the return value or a backfilled Status variable).

When the Xkb extension in the server hands an application a device identifier to use for the keyboard, that id is the input extension identifier for the device if the server supports the X Input Extension. If the server does not support the input extension, the meaning of the identifier is undefined — the only guarantee is that when you use XkbUseCoreKbd, XkbUseCoreKbd will work and the identifier returned by the server will refer to the core keyboard device.

3 Data Structures

An Xkb keyboard description consists of a variety of data structures, each of which describes some aspect of the keyboard. Although each data structure has its own peculiarities, there are a number of features common to nearly all Xkb structures. This chapter describes these common features and techniques for manipulating them.

Many Xkb data structures are interdependent; changing a field in one might require changes to others. As an additional complication, some Xkb library routines allocate related components as a group to reduce fragmentation and allocator overhead. In these cases, simply allocating and freeing fields of Xkb structures might corrupt program memory. Creating and destroying such structures or keeping them properly synchronized during editing is complicated and error prone.

Xkb provides functions and macros to allocate and free all major data structures. You should use them instead of allocating and freeing the structures yourself.

3.1 Allocating Xkb Data Structures

Xkb provides functions, known as allocators, to create and initialize Xkb data structures. In most situations, the Xkb functions that read a keyboard description from the server call these allocators automatically. As a result, you will seldom have to directly allocate or initialize Xkb data structures.

However, if you need to enlarge an existing structure or construct a keyboard definition from scratch, you may need to allocate and initialize Xkb data structures directly. Each major Xkb data structure has its own unique allocator. The allocator functions share common features: allocator functions for structures with optional components take as an input argument a mask of sub-components to be allocated. Allocators for data structures containing variable length data take an argument specifying the initial length of the data.

You may call an allocator to change the size of the space allocated for variable length data. When you call an allocator with an existing data structure as a parameter, the allocator does not change the data in any of the fields, with one exception: variable length data might be moved. The allocator resizes the allocated memory if the current size is too small. This normally involves allocating new memory, copying existing data to the newly allocated memory, and freeing the original memory. This possible reallocation is important to note because local variables pointing into Xkb data structures might be invalidated by calls to allocator functions.

3.2 Adding Data and Editing Data Structures

You should edit most data structures via the Xkb-supplied helper functions and macros, although a few data structures can be edited directly. The helper functions and macros make sure everything is initialized and interdependent values are properly updated for those Xkb structures that have interdependencies. As a general rule, if there is a helper function or macro to edit the data structure, use it. For example, increasing the width of a type requires you to resize every key that uses that type. This is complicated and ugly, which is why there's an *XkbResizeKeyType* function.

Many Xkb data structures have arrays whose size is reported by two fields. The first field, whose name is usually prefixed by *sz_*, represents the total number of elements that can be stored in the array. The second field, whose name is usually prefixed by *num_*, specifies

the number of elements currently stored there. These arrays typically represent data whose total size cannot always be determined when the array is created. In these instances, the usual way to allocate space and add data is as follows:

- Call the allocator function with some arbitrary size, as a hint.
- For those arrays which have an *Xkb...Add...* function, call it each time you want to add new data to the array. The function expands the array if necessary.

For example, call:

```
XkbAllocGeomShapes(geom,4)
```

to say “I’ll need space for four new shapes in this geometry.” This makes sure that $sz_shapes - num_shapes \geq 4$, and resizes the shapes array if it isn’t. If this function succeeds, you are guaranteed to have space for the number of shapes you need.

When you call an editing function for a structure, you do not need to check for space, because the function automatically checks the *sz_* and *num_* fields of the array, resizes the array if necessary, adds the entry to the array, and then updates the *num_* field.

3.3 Making Changes to the Server’s Keyboard Description

In Xkb, as in the core protocol, the client and server have independent copies of the data structures that describe the keyboard. The recommended way to change some aspect of the keyboard mapping in the X server is to edit a local copy of the Xkb keyboard description and then send only the changes to the X server. This method helps eliminate the need to transfer the entire keyboard description or even an entire data structure for only minor changes.

To help you keep track of the changes you make to a local copy of the keyboard description, Xkb provides separate special *changes* data structures for each major Xkb data structure. These data structures do not contain the actual changed values: they only indicate the changes which have been made to the structures that actually describe the keyboard.

When you wish to change the keyboard description in the server, you first modify a local copy of the keyboard description and then flag the modifications in an appropriate changes data structure. When you finish editing the local copy of the keyboard description, you pass your modified version of the keyboard description and the modified changes data structure to an Xkb routine. This routine uses the modified keyboard description and changes structure to pass only the changed information to the server. Note that modifying the keyboard description but not setting the appropriate flags in the changes data structure causes indeterminate behavior.

3.4 Tracking Keyboard Changes in the Server

The server reports all changes in its keyboard description to any interested clients via special Xkb events. Just as clients use special changes data structures to change the keyboard description in the server, the server uses special changes data structures to tell a client what changed in the server’s keyboard description.

Unlike clients, however, the server does not always pass the new values when it reports changes to its copy of the keyboard description. Instead, the server only passes a changes data structure when it reports changes to its keyboard description. This is done for efficiency reasons — some clients do not always need to update their copy of the keyboard description with every report from the server.

When your client application receives a report from the server indicating the keyboard description has changed, you can determine the set of changes by passing the event to an Xkb routine which “notes” event information in the corresponding changes data structure. These “note changes” routines are defined for all major Xkb components, and their names have the form *XkbNote{Component}Changes*, where *Component* is the name of a major Xkb component such as *Map* or *Names*. When you want to copy these changes from the server into a local copy of the keyboard description, call the corresponding *XkbGet{Component}Changes* function, passing it the changes structure. The function then retrieves only the changed structures from the server and copies the modified pieces into the local keyboard description.

3.5 Freeing Data Structures

For the same reasons you should not directly call *malloc* to allocate Xkb data structures, you should not free Xkb data structures or components directly using *free* or *Xfree*. Xkb provides functions to free the various data structures and their components. Always use the free functions supplied by Xkb. There is no guarantee that any particular field can be safely freed by *free* or *Xfree*.

4 Xkb Events

The primary way the X server communicates with clients is by sending X events to them. Some events are sent to all clients, while others are sent only to clients which have requested them. Some of the events that can be requested are associated with a particular window, and are only sent to those clients who have both requested the event and specified the window in which the event occurred.

The Xkb extension uses events to communicate the keyboard status to interested clients. These events are not associated with a particular window. Instead, all Xkb keyboard status events are reported to all interested clients, regardless of which window currently has the keyboard focus and regardless of the grab state of the keyboard.¹

The X server reports the events defined by the Xkb extension to your client application only if you have requested them. You may request Xkb events by calling either *XkbSelectEvents* or *XkbSelectEventDetails*. *XkbSelectEvents* requests Xkb events by their event type, and causes them to be reported to your client application under all circumstances. You can specify a finer granularity for event reporting by using *XkbSelectEventDetails*; in this case events are reported only when the specific detail conditions you specify have been met.

4.1 Xkb Event Types

The Xkb Extension adds new event types to the X protocol definition. An Xkb event type is defined by two fields in the X event data structure. One is the *type* field, containing the *base event code*. This base event code is a value the X server assigns to each X extension at runtime and which identifies the extension that generated the event; thus, the event code in the *type* field identifies the event as an Xkb extension event, rather than an event from another extension or a core X protocol event. You can obtain the base event code via a call to *XkbQueryExtension* or *XkbOpenDisplay*. The second field is the Xkb event type, which contains a value uniquely identifying each different Xkb event type. Possible values are defined by constants declared in the header file *<X11/extensions/Xkb.h>*.

Table 4.1 lists the categories of events defined by Xkb and their associated event types, as defined in *Xkb.h*. Each event is described in more detail in the section referenced for that event.

Table 4.1 Xkb Event Types

Event Type	Conditions Generating Event	Section	Page
XkbNewKeyboardNotify	Keyboard geometry; keycode range change	19	187
XkbMapNotify	Keyboard mapping change	14.4	122
XkbStateNotify	Keyboard state change	5.4	25
XkbControlsNotify	Keyboard controls state change	10.11	79
XkbIndicatorStateNotify	Keyboard indicators state change	8.5	45
XkbIndicatorMapNotify	Keyboard indicators map change	8.5	45
XkbNamesNotify	Keyboard name change	18.5	185
XkbCompatMapNotify	Keyboard compatibility map change	17.5	178
XkbBellNotify	Keyboard bell generated	9.4	52

1. The one exception to this rule is the *XkbExtensionDeviceNotify* event report which is sent when a client attempts to use an unsupported feature of an X Input Extension device (see section 21.4).

Table 4.1 Xkb Event Types

<i>Event Type</i>	<i>Conditions Generating Event</i>	<i>Section</i>	<i>Page</i>
XkbActionMessage	Keyboard action message	16.1.11	155
XkbAccessXNotify	AccessX state change	10.6.4	65
XkbExtensionDeviceNotify	Extension device change	21.6	207

4.2 Xkb Event Data Structures

Xkb reports each event it generates in a unique structure holding the data values needed to describe the conditions the event is reporting. However, all Xkb events have certain things in common. These common features are contained in the same fields at the beginning of all Xkb event structures, and are described in the `XkbAnyEvent` structure:

```
typedef struct {
    int           type;          /* Xkb extension base event code */
    unsigned long serial;        /* X server serial number for event */
    Bool          send_event;    /* True => synthetically generated */
    Display *     display;       /* server connection where event generated */
    Time          time;          /* server time when event generated */
    int           xkb_type;      /* Xkb minor event code */
    unsigned int   device;        /* Xkb device id, will not be XkbUseCoreKbd */
} XkbAnyEvent;
```

For any Xkb event, the *type* field is set to the base event code for the Xkb extension, assigned by the server to all Xkb extension events. The *serial*, *send_event*, and *display* fields are as described for all X11 events. The *time* field is set to the time when the event was generated and is expressed in milliseconds. The *xkb_type* field contains the minor extension event code, which is the extension event type, and is one of the values listed in Table 4.1. The *device* field contains the keyboard device identifier associated with the event. This is never `XkbUseCoreKbd`, even if the request that generated the event specified a device of `XkbUseCoreKbd`. If the request that generated the event specified `XkbUseCoreKbd`, *device* contains a value assigned by the server to specify the core keyboard. If the request that generated the event specified an X input extension device, *device* contains that same identifier.

Other data fields specific to individual Xkb events are described in subsequent chapters where the events are described.

4.3 Selecting Xkb Events

Xkb events are selected using an event mask, much the same as normal core X events are selected. However, unlike selecting core X events, where you must specify the selection status (on or off) for all possible event types whenever you wish to change the selection criteria for any one event, Xkb allows you to restrict the specification to only the event types you wish to change. This means you don't need to remember the event selection values for all possible types each time you want to change one of them.

Many Xkb event types are generated under several different circumstances. When selecting to receive an Xkb event, you may specify either that you want it delivered under all circumstances, or that you want it delivered only for a subset of the possible circumstances.

You can also deselect an event type that was previously selected for, using the same granularity.

Xkb provides two functions to select and deselect delivery of Xkb events. *XkbSelectEvents* allows you to select or deselect delivery of more than one Xkb event type at once. Events selected using *XkbSelectEvents* are delivered to your program under all circumstances which generate the events. To restrict delivery of an event to a subset of the conditions under which it occurs, call *XkbSelectEventDetails*. *XkbSelectEventDetails* only allows you to change the selection conditions for a single event at a time, but it provides a means of fine-tuning the conditions under which the event is delivered.

To select and / or deselect for delivery of one or more Xkb events and have them delivered under all conditions, call *XkbSelectEvents*.

```
Bool XkbSelectEvents(display, device_spec, bits_to_change, values_for_bits)
    Display *          display;           /* connection to the X server */
    unsigned int       device_spec;      /* device id, or XkbUseCoreKbd */
    unsigned long int bits_to_change; /* determines events to be selected / deselected */
    unsigned long int values_for_bits; /* 1=>select, 0->deselect; for events in bits_to_change */
```

This request changes the Xkb event selection mask for the keyboard specified by *device_spec*.

Each Xkb event that can be selected is represented by a bit in the *bits_to_change* and *values_for_bits* masks. Only the event selection bits specified by the *bits_to_change* parameter are affected; any unspecified bits are left unchanged. To turn on event selection for an event, set the bit for the event in the *bits_to_change* parameter and set the corresponding bit in the *values_for_bits* parameter. To turn off event selection for an event, set the bit for the event in the *bits_to_change* parameter and do not set the corresponding bit in the *values_for_bits* parameter. The valid values for both of these parameters are an inclusive bitwise OR of the masks shown in Table 4.2 on page 18. There is no interface to return your client's current event selection mask. Clients cannot set other clients' event selection masks.

If a bit is not set in the *bits_to_change* parameter, but the corresponding bit is set in the *values_for_bits* parameter, a BadMatch protocol error results. If an undefined bit is set in either the *bits_to_change* or the *values_for_bits* parameter, a BadValue protocol error results.

All event selection bits are initially zero for clients using the Xkb extension. Once you set some bits, they remain set for your client until you clear them via another call to *XkbSelectEvents*.

XkbSelectEvents returns False if the Xkb extension has not been initialized and True otherwise.

To select or deselect for a specific Xkb event and optionally place conditions on when events of that type are reported to your client, call *XkbSelectEventDetails*. This allows you

to exercise a finer granularity of control over delivery of Xkb events with *XkbSelectEvents*.

```
Bool XkbSelectEventDetails(display, device_spec, event_type, bits_to_change, values_for_bits)
    Display *      display;          /* connection to the X server */
    unsigned int   device_spec;     /* device id, or XkbUseCoreKbd */
    unsigned int   event_type;      /* Xkb event type of interest */
    unsigned long int bits_to_change; /* event selection details */
    unsigned long int values_for_bits; /* values for bits selected by bits_to_change */
```

While *XkbSelectEvents* allows multiple events to be selected, *XkbSelectEventDetails* changes the selection criteria for a single type of Xkb event. The interpretation of the *bits_to_change* and *values_for_bits* masks depends on the event type in question.

XkbSelectEventDetails changes the Xkb event selection mask for the keyboard specified by *device_spec* and the Xkb event specified by *event_type*. To turn on event selection for an event detail, set the bit for the detail in the *bits_to_change* parameter and set the corresponding bit in the *values_for_bits* parameter. To turn off event detail selection for an detail, set the bit for the detail in the *bits_to_change* parameter and do not set the corresponding bit in the *values_for_bits* parameter.

If an invalid event type is specified, a *BadValue* protocol error results. If a bit is not set in the *bits_to_change* parameter, but the corresponding bit is set in the *values_for_bits* parameter, a *BadMatch* protocol error results. If an undefined bit is set in either the *bits_to_change* or the *values_for_bits* parameter, a *BadValue* protocol error results.

For each type of Xkb event, the legal event details that you can specify in the *XkbSelectEventDetails* request are listed in the chapters that describe each event in detail.

4.3.1 Event Masks

The X server reports the events defined by Xkb to your client application only if you have requested them via a call to *XkbSelectEvents* or *XkbSelectEventDetails*. Specify the event types in which you are interested in a mask, as described in section 4.3.

Table 4.2 lists the event mask constants that can be specified with the *XkbSelectEvents* request, and the circumstances in which the mask should be specified.

Table 4.2 XkbSelectEvents Mask Constants

<i>Event Mask</i>	<i>Value</i>	<i>Notification Wanted</i>
XkbNewKeyboardNotifyMask	(1L<<0)	Keyboard geometry change
XkbMapNotifyMask	(1L<<1)	Keyboard mapping change
XkbStateNotifyMask	(1L<<2)	Keyboard state change
XkbControlsNotifyMask	(1L<<3)	Keyboard control change
XkbIndicatorStateNotifyMask	(1L<<4)	Keyboard indicator state change
XkbIndicatorMapNotifyMask	(1L<<5)	Keyboard indicator map change
XkbNamesNotifyMask	(1L<<6)	Keyboard name change
XkbCompatMapNotifyMask	(1L<<7)	Keyboard compat map change
XkbBellNotifyMask	(1L<<8)	Bell
XkbActionMessageMask	(1L<<9)	Action message
XkbAccessXNotifyMask	(1L<<10)	AccessX features
XkbExtensionDeviceNotifyMask	(1L<<11)	Extension device

Table 4.2 XkbSelectEvents Mask Constants

<i>Event Mask</i>	<i>Value</i>	<i>Notification Wanted</i>
XkbAllEventsMask	(0xFFFF)	All Xkb events

4.4 Unified Xkb Event Type

The `XkbEvent` structure is a union of the individual structures declared for each Xkb event type and for the core protocol `XEvent` type. Given an `XkbEvent` structure, you may use the `type` field to determine if the event is an Xkb event (`type` equals the Xkb base event code; see section 2.4). If the event is an Xkb event, you may then use the `any.xkb_type` field to determine the type of Xkb event, and thereafter access the event dependent components using the union member corresponding to the particular Xkb event type.

```
typedef union _XkbEvent {
    int type;
    XkbAnyEvent any;
    XkbStateNotifyEvent state;
    XkbMapNotifyEvent map;
    XkbControlsNotifyEvent ctrls;
    XkbIndicatorNotifyEvent indicators;
    XkbBellNotifyEvent bell;
    XkbAccessXNotifyEvent accessx;
    XkbNamesNotifyEvent names;
    XkbCompatMapNotifyEvent compat;
    XkbActionMessageEvent message;
    XkbExtensionDeviceNotifyEvent device;
    XkbNewKeyboardNotifyEvent new_kbd;
    XEvent core;
} XkbEvent;
```

This unified Xkb event type includes a normal `XEvent` as used by the core protocol, so it is straightforward for applications that use Xkb events to call the X library event functions without having to cast every reference. For example, to get the next event, you can simply declare a variable of type `XkbEvent` and call:

```
XNextEvent(dpy,&xkbев.core);
```

5 Keyboard State

Keyboard state encompasses all of the transitory information necessary to map a physical key press or release to an appropriate event. The Xkb keyboard state consists of primitive components and additional derived components which are maintained for efficiency reasons. The diagram below shows the components of Xkb keyboard state and their relationships.

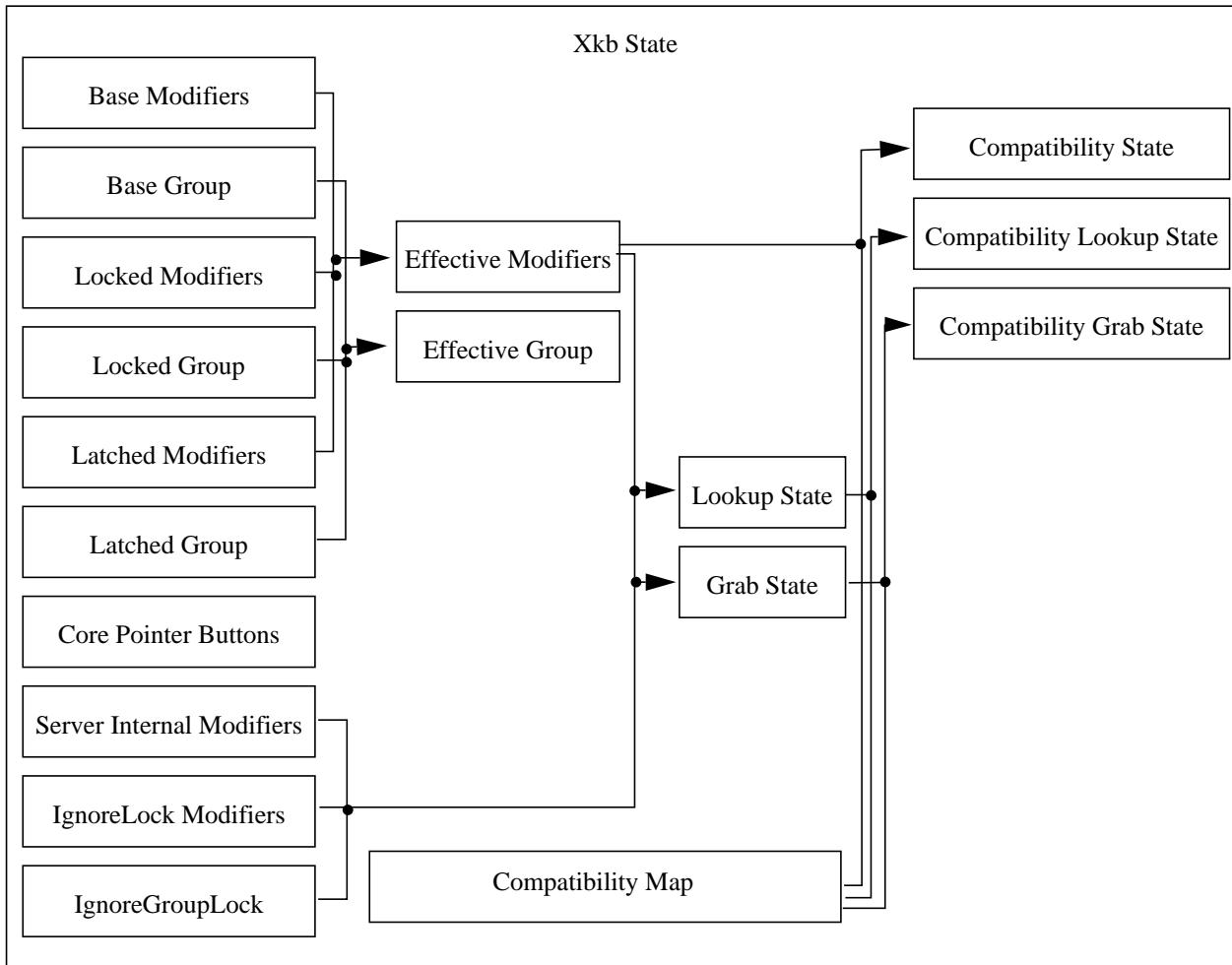


Figure 5.1 Xkb State

5.1 Keyboard State Description

The Xkb keyboard state is comprised of the state of all keyboard modifiers, the keyboard group, and the state of the pointer buttons. These are grouped into the following components:

- The locked group and locked modifiers.
- The latched group and latched modifiers.
- The base group and base modifiers.
- The effective group and effective modifiers.
- The state of the core pointer buttons.

The *modifiers* are Shift, Lock, Control, and Mod1-Mod5, as defined by the core protocol. A modifier can be thought of as a toggle which is either set or unset. All modifiers are initially unset. When a modifier is locked, it is set and remains set for all future key events, until it is explicitly unset. A latched modifier is set, but automatically unsets after the next key event that does not change the keyboard state. Locked and latched modifier state can be changed by keyboard activity or via Xkb extension library functions.

The Xkb extension provides support for *keysym groups*, as defined by ISO9995:

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.
-------	---

The Xkb extension supports up to four keysym groups. Groups are named beginning with one and indexed beginning with zero. All group states are indicated using the group index. At any point in time, there is zero or one locked group, zero or one latched group, and one base group. When a group is locked, it supersedes any previous locked group and remains the locked group for all future key events, until a new group is locked. A latched group applies only to the next key event that does not change the keyboard state. The locked and latched group can be changed by keyboard activity or via Xkb extension library functions.

Changing to a different group changes the keyboard state to produce characters from a different group. Groups are typically used to switch between keysyms of different languages and locales.

The *pointer buttons* are Button1 - Button5, as defined by the core protocol.

The *base group* and *base modifiers* represent keys that are physically or logically down. These and the pointer buttons can be changed by keyboard activity and not by Xkb requests. It is possible for a key to be logically down, but not physically down, and neither latched nor locked.¹

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

The *effective group* is the arithmetic sum of the group indices of the latched group, locked group, and base group, which is then normalized by some function. The result is a meaningful group index.

$$\begin{aligned} n &= \text{number of keyboard groups, } 1 \leq n \leq 4 \\ 0 &\leq \text{any of locked, latched, or base group} < n \\ \text{effective group} &= f(\text{locked group} + \text{latched group} + \text{base group}) \end{aligned}$$

The function f insures the effective group is within range. The precise function is specified for the keyboard and can be retrieved through the keyboard description. It may wrap around, clamp down, or default. Few applications will actually examine the effective group and far fewer still will examine the locked, latched, and base groups.

There are two circumstances under which groups are normalized:

1. Keys may be logically down when they are physically up due to their electrical properties, or due to the keyboard extension in the X server having filtered the key release, for esoteric reasons.

1. The global locked or effective group changes. In this case, the changed group is normalized into range according to the settings of the *groups_wrap* field of the `XkbControlsRec` structure for the keyboard (see section 10.7.1).
2. The Xkb library is interpreting an event with an effective group that is legal for the keyboard as a whole, but not for the key in question. In this case, the group to use for this event only is determined using the *group_info* field of the key symbol mapping (`XkbSymMapRec`) for the event key.

Each non-modifier key on a keyboard has zero or more symbols, or keysyms, associated with it. These are the logical symbols which the key can generate when it is pressed. The set of all possible keysyms for a keyboard is divided into groups. Each key is associated with zero or more groups; each group contains one or more symbols. When a key is pressed, the determination of which symbol for the key is selected is based on the effective group and the shift level, which is determined by which modifiers are set.

A client which does not explicitly call Xkb functions, but which otherwise makes use of an X library containing the Xkb extension, will have keyboard state represented in bits 0 - 14 of the state field of events which report modifier and button state. Such a client is said to be *Xkb-capable*. A client which does explicitly call Xkb functions is an *Xkb-aware* client. The Xkb keyboard state includes information derived from the effective state and from two server parameters which can be set through the keyboard extension. The following components of keyboard state pertain to Xkb-capable and Xkb-aware clients:

- lookup state: lookup group and lookup modifiers
- grab state: grab group and grab modifiers

The *lookup modifiers* and *lookup group* are represented in the state field of core X events. The modifier state and keycode of a key event are used to determine the symbols associated with the event. For KeyPress and KeyRelease events, the lookup modifiers are computed as:

$((base \mid latched \mid locked) \& \sim server_internal_modifiers)$

Otherwise the lookup modifiers are computed as:

$((base \mid latched \mid (locked \& \sim ignore_locks)) \& \sim server_internal_modifiers)$

The lookup group is the same as the effective group.

When an Xkb-capable or Xkb-aware client wishes to map a keycode to a keysym, it should use the *lookup state* — the lookup group and the lookup modifiers.

The *grab state* is the state used when matching events to passive grabs. If the event activates a grab, the *grab modifiers* and *grab group* are represented in the state field of core X events; otherwise, the lookup state is used. The grab modifiers are computed as:

$((base \mid latched \mid (locked \& \sim ignore_locks)) \& \sim server_internal_modifiers)$

If the server's `IgnoreGroupLock` control (see section 10.7.3) is not set, the grab group is the same as the effective group. Otherwise, the grab group is computed from the base group and latched group, ignoring the locked group.

The final three components of Xkb state are applicable to clients which are not linked with an Xlib containing the X keyboard extension library and therefore are not aware of the keyboard extension (*Xkb-unaware* clients):

- The compatibility modifier state.
- The compatibility lookup modifier state.
- The compatibility grab modifier state.

The X11 protocol interpretation of modifiers does not include direct support for multiple groups. When an Xkb-extended X server connects to an Xkb-unaware client, the compatibility states remap the keyboard group into a core modifier whenever possible. The compatibility state corresponds to the effective modifier and effective group state, with the group remapped to a modifier. The compatibility lookup and grab states correspond to the lookup and grab states, respectively, with the group remapped to a modifier. The compatibility lookup state is reported in events that do not trigger passive grabs, otherwise the compatibility grab state is reported.

5.2 Changing the Keyboard State

5.2.1 Changing Modifiers

The functions in this section which change the use of modifiers use a mask in the parameter *affect*. It is a bitwise inclusive OR of the legal modifier masks:

Table 5.1 Real Modifier Masks

<i>Mask</i>
ShiftMask
LockMask
ControlMask
Mod1Mask
Mod2Mask
Mod3Mask
Mod4Mask
Mod5Mask

To lock and unlock any of the eight real keyboard modifiers, use *XkbLockModifiers*:

```
Bool XkbLockModifiers(display, device_spec, affect, values)
    Display * display; /* connection to the X server */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int affect; /* mask of real modifiers whose lock state is to change */
    unsigned int values; /* 1 => lock, 0 => unlock; only for modifiers selected by affect */
```

XkbLockModifiers sends a request to the server to lock the real modifiers selected by both *affect* and *values*, and to unlock the real modifiers selected by *affect* but not selected by *values*. *XkbLockModifiers* does not wait for a reply from the server. It returns True if the request was sent, and False otherwise.

To latch and unlatch any of the eight real keyboard modifiers, use *XkbLatchModifiers*:

```
Bool XkbLatchModifiers(display, device_spec, affect, values)
    Display * display; /* connection to the X server */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int affect; /* mask of modifiers whose latch state is to change */
    unsigned int values; /* 1 => latch, 0 => unlatch; only for mods selected by affect */
```

XkbLatchModifiers sends a request to the server to latch the real modifiers selected by both *affect* and *values*, and to unlatch the real modifiers selected by *affect* but not selected by *values*. *XkbLatchModifiers* does not wait for a reply from the server. It returns True if the request was sent, and False otherwise.

5.2.2 Changing Groups

Reference the keysym group indices with these symbolic constants:

Table 5.2 Symbolic Group Names

<i>Symbolic Name</i>	<i>Value</i>
XkbGroup1Index	0
XkbGroup2Index	1
XkbGroup3Index	2
XkbGroup4Index	3

To lock the keysym group, call *XkbLockGroup*.

```
Bool XkbLockGroup(display, device_spec, group)
    Display *    display;      /* connection to the X server */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int group;       /* index of the keysym group to lock */
```

XkbLockGroup sends a request to the server to lock the specified *group*, and does not wait for a reply. It returns True if the request was sent, and False otherwise.

To latch the keysym group, call *XkbLatchGroup*.

```
Bool XkbLatchGroup(display, device_spec, group)
    Display *    display;      /* connection to the X server */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int group;       /* index of the keysym group to latch */
```

XkbLatchGroup sends a request to the server to latch the specified group, and does not wait for a reply. It returns True if the request was sent, and False otherwise.

5.3 Determining Keyboard State

Xkb keyboard state may be represented in an *XkbStateRec* structure, defined below:

```
typedef struct {
    unsigned char group;           /* effective group index */
    unsigned char base_group;      /* base group index */
    unsigned char latched_group;   /* latched group index */
    unsigned char locked_group;    /* locked group index */
    unsigned char mods;            /* effective modifiers */
    unsigned char base_mods;       /* base modifiers */
    unsigned char latched_mods;    /* latched modifiers */
    unsigned char locked_mods;     /* locked modifiers */
    unsigned char compat_state;    /* effective group => modifiers */
    unsigned char grab_mods;       /* modifiers used for grabs */
    unsigned char compat_grab_mods; /* mods used for compatibility mode grabs */
    unsigned char lookup_mods;     /* modifiers used to lookup symbols */
    unsigned char compat_lookup_mods; /* mods used for compatibility lookup */
```

```

    unsigned short  ptr_buttons;          /* 1 bit => corresponding pointer btn is down */
} XkbStateRec,*XkbStatePtr;

```

To obtain the keyboard state, use *XkbGetState*.

```

Status XkbGetState(display, device_spec, state_return)
Display *           display;        /* connection to the X server */
unsigned int        device_spec;   /* device id, or XkbUseCoreKbd */
XkbStatePtr         state_return;  /* backfilled with Xkb state */

```

The *XkbGetState* function queries the server for the current keyboard state, waits for a reply, and then backfills *state_return* with the results.

All group values are expressed as group indices in the range [0..3]. Modifiers and the compatibility modifier state values are expressed as the bitwise union of the core X11 modifier masks. The pointer button state is reported as in the core X11 protocol.

5.4 Tracking Keyboard State

The Xkb extension reports *XkbStateNotify* events to clients wanting notification whenever the Xkb state changes. The changes reported include changes to any aspect of the keyboard state: when a modifier is set or unset, when the current group changes, or when a pointer button is pressed or released. As with all Xkb events, *XkbStateNotify* events are reported to all interested clients without regard to the current keyboard input focus or grab state.

There are many different types of Xkb state changes. Xkb defines an event detail mask corresponding to each type of change. The event detail masks are listed in Table 5.3.

Table 5.3 XkbStateNotify Event Detail Masks

Mask	Value
XkbModifierStateMask	(1L << 0)
XkbModifierBaseMask	(1L << 1)
XkbModifierLatchMask	(1L << 2)
XkbModifierLockMask	(1L << 3)
XkbGroupStateMask	(1L << 4)
XkbGroupBaseMask	(1L << 5)
XkbGroupLatchMask	(1L << 6)
XkbGroupLockMask	(1L << 7)
XkbCompatStateMask	(1L << 8)
XkbGrabModsMask	(1L << 9)
XkbCompatGrabModsMask	(1L << 10)
XkbLookupModsMask	(1L << 11)
XkbCompatLookupModsMask	(1L << 12)
XkbPointerButtonMask	(1L << 13)
XkbAllStateComponentsMask	(0x3fff)

To track changes in the keyboard state for a particular device, select to receive *XkbStateNotify* events by calling either *XkbSelectEvents* or *XkbSelectEventDetails* (see section 4.3).

To receive `XkbStateNotify` events under all possible conditions, call `XkbSelectEvents` and pass `XkbStateNotifyMask` in both `bits_to_change` and `values_for_bits`.

To receive `XkbStateNotify` events only under certain conditions, call `XkbSelectEventDetails` using `XkbStateNotify` as the `event_type` and specifying the desired state changes in `bits_to_change` and `values_for_bits` using mask bits from Table 5.3.

The structure for `XkbStateNotify` events is:

```
typedef struct {
    int             type;          /* Xkb extension base event code */
    unsigned long   serial;        /* X server serial number for event */
    Bool            send_event;    /* True => synthetically generated */
    Display *      display;       /* server connection where event generated */
    Time           time;          /* server time when event generated */
    int             xkb_type;     /* XkbStateNotify */
    int             device;        /* Xkb device id, will not be XkbUseCoreKbd */
    unsigned int    changed;       /* bits indicating what has changed */
    int             group;         /* group index of effective group */
    int             base_group;    /* group index of base group */
    int             latched_group; /* group index of latched group */
    int             locked_group;  /* group index of locked group */
    unsigned int    mods;          /* effective modifiers */
    unsigned int    base_mods;     /* base modifiers */
    unsigned int    latched_mods;  /* latched modifiers */
    unsigned int    locked_mods;   /* locked modifiers */
    int             compat_state;  /* computed compatibility state */
    unsigned char   grab_mods;    /* modifiers used for grabs */
    unsigned char   compat_grab_mods; /* modifiers used for compatibility grabs */
    unsigned char   lookup_mods;   /* modifiers used to lookup symbols */
    unsigned char   compat_lookup_mods; /* mods used for compatibility look up */
    int             ptr_buttons;   /* core pointer buttons */
    KeyCode         keycode;       /* keycode causing event, 0 if programmatic */
    char            event_type;    /* core event if req_major or req_minor non zero */
    char            req_major;     /* major request code if program trigger, else 0 */
    char            req_minor;     /* minor request code if program trigger, else 0 */
} XkbStateNotifyEvent;
```

When you receive an `XkbStateNotify` event, the `changed` field indicates which elements of keyboard state have changed. This will be the bitwise inclusive OR of one or more of the `XkbStateNotify` event detail masks shown in Table 5.3 on page 25. All fields reported in the event are valid, but only those indicated in `changed` have changed values.

The `group` field is the group index of the effective keysym group. The `base_group`, `latched_group`, and `locked_group` fields are set to a group index value representing the base group, the latched group, and the locked group, respectively. The X server can set the modifier and compatibility state fields to a union of the core modifier mask bits; this union represents the corresponding modifier states. The `ptr_button` field gives the state of the core pointer buttons as a mask composed of an inclusive OR of zero or more of the core pointer button masks.

Xkb state changes can occur either in response to keyboard activity or under application control. If a key event caused the state change, the *keycode* field gives the keycode of the key event, and the *event_type* field is set to either KeyPress or KeyRelease. If a pointer button event caused the state change, the *keycode* field is zero, and the *event_type* field is set to either ButtonPress or ButtonRelease. Otherwise, the major and minor codes of the request that caused the state change are given in the *req_major* and *req_minor* fields, and the *keycode* field is zero. The *req_major* value is the same as the major extension opcode.

6 Complete Keyboard Description

The complete Xkb description for a keyboard device is accessed using a single structure containing pointers to major Xkb components. This chapter describes this single structure and provides references to other sections of this document that discuss the major Xkb components in detail.

6.1 The XkbDescRec Structure

The complete description of an Xkb keyboard is given by an `XkbDescRec`. The component structures in the `XkbDescRec` represent the major Xkb components outlined in Figure 1.1 on page 3.

```
typedef struct {
    struct _XDisplay *display;      /* connection to X server */
    unsigned short flags;          /* private to Xkb, do not modify */
    unsigned short device_spec;    /* device of interest */
    KeyCode min_key_code;          /* minimum keycode for device */
    KeyCode max_key_code;          /* maximum keycode for device */
    XkbControlsPtr ctrls;         /* controls */
    XkbServerMapPtr server;        /* server keymap */
    XkbClientMapPtr map;          /* client keymap */
    XkbIndicatorPtr indicators;   /* indicator map */
    XkbNamesPtr names;            /* names for all components */
    XkbCompatMapPtr compat;        /* compatibility map */
    XkbGeometryPtr geom;          /* physical geometry of keyboard */
} XkbDescRec, *XkbDescPtr;
```

The `display` field points to an X display structure. The `flags` field is private to the library: modifying `flags` may yield unpredictable results. The `device_spec` field specifies the device identifier of the keyboard input device, or `XkbUseCoreKeyboard`, which specifies the core keyboard device. The `min_key_code` and `max_key_code` fields specify the least and greatest keycode that can be returned by the keyboard.

The other fields specify structure components of the keyboard description and are described in detail in other sections of this document. Table 6.1 identifies the subsequent sections of this document which discuss the individual components of the `XkbDescRec`.

Table 6.1 XkbDescRec Component References

<i>XkbDescRec Field</i>	<i>For more info</i>
ctrls	Chapter 10
server	Chapter 16
map	Chapter 15
indicators	Chapter 8
names	Chapter 18
compat	Chapter 17
geom	Chapter 13

Each structure component has a corresponding mask bit which is used in function calls to indicate that the structure should be manipulated in some manner, such as allocating it or

freeing it. These masks and their relationships to the fields in the `XkbDescRec` are shown in Table 6.2.

Table 6.2 Mask Bits for `XkbDescRec`

<i>Mask Bit</i>	<i>XkbDescRec Field</i>	<i>Value</i>
<code>XkbControlsMask</code>	<code>ctrls</code>	<code>(1L<<0)</code>
<code>XkbServerMapMask</code>	<code>server</code>	<code>(1L<<1)</code>
<code>XkbIClientMapMask</code>	<code>map</code>	<code>(1L<<2)</code>
<code>XkbIndicatorMapMask</code>	<code>indicators</code>	<code>(1L<<3)</code>
<code>XkbNamesMask</code>	<code>names</code>	<code>(1L<<4)</code>
<code>XkbCompatMapMask</code>	<code>compat</code>	<code>(1L<<5)</code>
<code>XkbGeometryMask</code>	<code>geom</code>	<code>(1L<<6)</code>
<code>XkbAllComponentsMask</code>	All Fields	<code>(0x7f)</code>

6.2 Obtaining a Keyboard Description from the Server

To retrieve one or more components of a keyboard device description, use `XkbGetKeyboard` (see also `XkbGetKeyboardByName`, on page 194).

`XkbDescPtr XkbGetKeyboard(display, which, device_spec)`

```
Display * display; /* connection to X server */
unsigned int which; /* mask indicating components to return */
unsigned int device_spec; /* device for which to fetch description, or XkbUseCoreKbd */
```

`XkbGetKeyboard` allocates and returns a pointer to a keyboard description. It queries the server for those components specified in the `which` parameter for device `device_spec` and copies the results to the `XkbDescRec` it allocated. The remaining fields in the keyboard description are set to NULL. The valid masks for `which` are those listed in Table 6.2.

`XkbGetKeyboard` can generate `BadAlloc` protocol errors.

To free the returned keyboard description, use `XkbFreeKeyboard` (see section 6.4).

6.3 Tracking Changes to the Keyboard Description in the Server

The server can generate events whenever its copy of the keyboard description for a device changes. Refer to section 14.4 on page 122 for detailed information on tracking changes to the keyboard description.

6.4 Allocating and Freeing a Keyboard Description

Applications seldom need to directly allocate a keyboard description; calling `XkbGetKeyboard` usually suffices. In the event you need to create a keyboard description from scratch, however, use `XkbAllocKeyboard` rather than directly calling `malloc` or `Xmalloc`.

`XkbDescRec * XkbAllocKeyboard(void)`

If `XkbAllocKeyboard` fails to allocate the keyboard description, it returns `NULL`. Otherwise, it returns a pointer to an empty keyboard description structure. The `device_spec` field will have been initialized to `XkbUseCoreKbd`. You may then either fill in the structure components or use Xkb routines to obtain values for the structure components from a keyboard device.

To destroy either an entire an `XkbDescRec` or just some of its members, use `XkbFreeKeyboard`.

```
void XkbFreeKeyboard(xkb, which, free_all)
    XkbDescPtr      xkb;          /* keyboard description with components to free */
    unsigned int     which;        /* mask selecting components to free */
    Bool            free_all;     /* True => free all components and xkb */
```

`XkbFreeKeyboard` frees the components of `xkb` specified by `which` and sets the corresponding values to `NULL`. If `free_all` is `True`, `XkbFreeKeyboard` frees every non-`NULL` component of `xkb` and then frees the `xkb` structure itself.

7 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 the Num_Lock keysym is bound to some modifier, that modifier is used to select between shifted and 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. Using the core protocol only, a client application 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. It must repeat this process for all modifiers whenever any part of the modifier mapping is changed.

Xkb alleviates these problems by defining virtual modifiers. In addition to the eight core modifiers, referred to as the *real modifiers*, Xkb provides a set of sixteen named *virtual modifiers*. Each virtual modifier can be bound to any set of the real modifiers (Shift, Lock, Control and Mod1-Mod5).

The separation of function from physical modifier bindings 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 can use virtual modifiers to specify the desired behavior, without regard for the actual physical bindings in effect.

7.1 Virtual Modifier Names and Masks

Virtual modifiers are named by converting their string name to an X Atom and storing the Atom in the *names.vmods* array in an XkbDescRec structure (see section 6.1). The position of a name Atom in the *names.vmods* array defines the bit position used to represent the virtual modifier, and also the index used when accessing virtual modifier information in arrays: the name in the i-th (0 relative) entry of *names.vmods* is the i-th virtual modifier, represented by the mask (1<<i). Throughout Xkb, various functions have a parameter which is a mask representing virtual modifier choices. In each case, the i-th bit (0 relative) of the mask represents the i-th virtual modifier.

To set the name of a virtual modifier, call *XkbSetNames*, using *XkbVirtualModName-Mask* in *which* and the name in the *xb* argument; to retrieve indicator names, call *XkbGetNames*. These functions are discussed in Chapter 18.

7.2 Modifier Definitions

An Xkb *modifier definition* enumerates a collection of real and virtual modifiers but does not in itself bind those modifiers to any particular key or to each other. Modifier definitions are included in a number of structures in the keyboard description to define the collection of modifiers that affect or are affected by some other entity. A modifier definition is only relevant in the context of some other entity such as an indicator map, a control, or a key type. (See sections 8.2.2, 10.8 and 15.2.)

```
typedef struct _XkbMods {
    unsigned char      mask;          /* real_mods | vmods mapped to real modifiers */
    unsigned char      real_mods;     /* real modifier bits */
```

```
unsigned short      vmods;      /* virtual modifier bits */
} XkbModsRec,*XkbModsPtr;
```

An Xkb modifier definition consists of a set of bit masks corresponding to the eight real modifiers (*real_mods*); a similar set of bitmasks corresponding to the 16 named virtual modifiers (*vmods*); and an effective mask (*mask*). The effective mask represents the set of all real modifiers which can logically be set either by setting any of the real modifiers or by setting any of the virtual modifiers in the definition. *mask* is derived from the real and virtual modifiers and should never be explicitly changed — it contains all of the real modifiers specified in the definition (*real_mods*) *plus* any real modifiers that are bound to the virtual modifiers specified in the definition (*vmods*). The binding of the virtual modifiers to real modifiers is exterior to the modifier definition. Xkb automatically recomputes the mask field of modifier definitions as necessary. Whenever you access a modifier definition which has been retrieved using an Xkb library function, the mask field will be correct for the keyboard mapping of interest.

7.3 Binding Virtual Modifiers to Real Modifiers

The binding of virtual modifiers to real modifiers is defined by the *server.vmods* array in an XkbDescRec structure. Each entry contains the real modifier bits which are bound to the virtual modifier corresponding to the entry. The overall relationship of fields dealing with virtual modifiers in the server keyboard description are shown in Figure 16.2 on page 165.

7.4 Virtual Modifier Key Mapping

Xkb maintains a *virtual modifier mapping*, which lists the virtual modifiers associated with, or bound to, 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. The *server.vmodmap* array indicates which virtual modifiers are bound to each key; each entry is a bitmask for the virtual modifier bits. The *server.vmodmap* array is indexed by keycode.

The *vmodmap* and *vmods* members of the server map are the “master” virtual modifier definitions. Xkb automatically propagates any changes to these fields to all other fields that use virtual modifier mappings (see section 16.4 on page 164).

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 NumLock.

The virtual modifier mapping is normally updated whenever actions are automatically applied to symbols (see section 16.4 for details) and few applications should need to change the virtual modifier mapping explicitly.

Call *XkbGetMap* (see section 14.2) to get the virtual modifiers from the server or call *XkbGetVirtualMods* (see section 16.4.1) to update a local copy of the virtual modifiers bindings from the server. To set the binding of a virtual modifier to a real modifier, call *XkbSetMap* (see section 14.3).

To determine the mapping of virtual modifiers to core X protocol modifiers, use *XkbVirtualModsToReal*.

```
Bool XkbVirtualModsToReal(xkb, virtual_mask, mask_rtrn)
    XkbDescPtr           xkb;          /* keyboard description for input device */
    unsigned int         virtual_mask; /* virtual modifier mask to translate */
    unsigned int *       mask_rtrn; /* backfilled with real modifiers */
```

If the keyboard description defined by *xkb* includes bindings for virtual modifiers, *XkbVirtualModsToReal* uses those bindings to determine the set of real modifiers that correspond to the set of virtual modifiers specified in *virtual_mask*. The *virtual_mask* parameter is a mask specifying the virtual modifiers to translate; the *i*-th bit (0 relative) of the mask represents the *i*-th virtual modifier. If *mask_rtrn* is non-NULL, *XkbVirtualModsToReal* backfills it with the resulting real modifier mask. If the keyboard description in *xkb* does not include virtual modifier bindings, *XkbVirtualModsToReal* returns False, otherwise it returns True.

Note It is possible for a local (client-side) keyboard description (the *xkb* parameter) to not contain any virtual modifier information (simply because the client has not requested it) while the server's corresponding definition may contain virtual modifier information.

7.4.1 Inactive Modifier Sets

An unbound virtual modifier is one which is not bound to any real modifier (*server->vmods[virtual_modifier_index]* is zero).

Some Xkb operations 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, the effective masks for these two cases are identical (that is, 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.

7.5 Conventions

The Xkb extension does not require any specific virtual modifier names. However, everyone benefits if the same names are used for common modifiers. The following names are suggested:

```
NumLock
ScrollLock
Alt
Meta
AltGr
LevelThree
```

7.6 Example

If the 2nd (0-relative) entry in *names.vmods* contains the Atom for “NumLock”, then 0x4 (1<<2) is the virtual modifier bit for the NumLock virtual modifier. If *server.vmods[2]*

contains Mod3Mask, then the NumLock virtual modifier is bound to the Mod3 real modifier.

A virtual modifier definition for this example would have:

```
real_mods = 0
vmods = 0x4 (NumLock named virtual modifier)
mask = 0x20 (Mod3Mask)
```

Continuing the above example, if the keyboard has a Num_Lock keysym bound to the key with keycode 14, and the NumLock virtual modifier is bound to this key, *server.vmod-map[14]* contains 0x4.

Finally, if the keyboard also used the real Mod1 modifier for numeric lock operations, the modifier definition below would represent the situation where either the key bound to Mod1 or the NumLock virtual modifier could be used for this purpose:

```
real_mods = 0x8 (Mod1Mask)
vmods = 0x4 (NumLock named virtual modifier)
mask = 0x28 (Mod1Mask | Mod3Mask)
```

8 Indicators

Although the core X implementation supports up to 32 LEDs on an input device, it does not provide any linkage between the state of the LEDs and the logical state of the input device. For example, most keyboards have a CapsLock LED, but X does not provide a mechanism to make the LED automatically follow the logical state of the CapsLock key.

Furthermore, the core X implementation does not provide clients with the ability to determine what bits in the *led_mask* field of the `XKeyboardState` map to the particular LEDs on the keyboard. For example, X does not provide a method for a client to determine what bit to set in the *led_mask* field to turn on the Scroll Lock LED, or if the keyboard even has a Scroll Lock LED.

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. This chapter describes Xkb indicators and the routines used for manipulating them.

8.1 Indicator Names

Xkb provides the capability of symbolically naming indicators. Xkb itself doesn’t use these symbolic names for anything; they are only there to help make the keyboard description comprehensible to humans. To set the names of specific indicators, call `XkbSetNames` as discussed in Chapter 18. Then set the map using `XkbSetMap` (see section 14.3) or `XkbSetNamedIndicator` (below). To retrieve indicator names, call `XkbGetNames` (Chapter 18).

8.2 Indicator Data Structures

Use the indicator description record, `XkbIndicatorRec`, and its indicator map, `XkbIndicatorMapRec`, to inquire about and control most indicator properties and behaviors.

8.2.1 `XkbIndicatorRec`

The description for all the Xkb indicators is held in the *indicators* field of the complete keyboard description (see Chapter 6), which is defined as follows:

```
#define XkbNumIndicators      32
typedef struct {
    unsigned long          phys_indicators;           /* LEDs existence */
    XkbIndicatorMapRec     maps[XkbNumIndicators];    /* indicator maps */
} XkbIndicatorRec,*XkbIndicatorPtr;
```

This structure contains the *phys_indicators* field, which relates some information about the correspondence between indicators and physical LEDs on the keyboard, and an array of indicator *maps*, one map per indicator.

The *phys_indicators* field indicates which indicators are bound to physical LEDs on the keyboard; if a bit is set in *phys_indicators*, then the associated indicator has a physical

LED associated with it. This field is necessary because some indicators may not have corresponding physical LEDs on the keyboard. For example, most keyboards have an LED for indicating the state of CapsLock, but most keyboards do not have an LED that indicates the current group. Because *phys_indicators* describes a physical characteristic of the keyboard, you cannot directly change it under program control. However, if a client program loads a completely new keyboard description via *XkbGetKeyboardByName*, or if a new keyboard is attached and the X implementation notices, *phys_indicators* changes if the indicators for the new keyboard are different.

8.2.2 XkbIndicatorMapRec

Each indicator has its own set of attributes that specify if clients can explicitly set its state and if it tracks the keyboard state. The attributes of each indicator are held in the *maps* array, which is an array of *XkbIndicatorRec* structures:

```
typedef struct {
    unsigned char    flags;          /* how the indicator can be changed */
    unsigned char    which_groups;   /* match criteria for groups */
    unsigned char    groups;         /* which keyboard groups the indicator watches */
    unsigned char    which_mods;    /* match criteria for modifiers */
    XkbModsRec      mods;          /* which modifiers the indicator watches */
    unsigned int     ctrls;         /* which controls the indicator watches */
} XkbIndicatorMapRec, *XkbIndicatorMapPtr;
```

This indicator map specifies for each indicator:

- 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 functions *XkbSetControls* or *XChangeKeyboardControl*.

For more information on the effects of explicit changes to indicators and the relationship to the indicator map, see section 8.4.1 on page 42.

XkbIndicatorMapRec flags field

The *flags* field specifies the conditions under which the indicator can be changed and the effects of changing the indicator. The valid values for *flags* and their effects are shown in Table 8.1.

Table 8.1 XkbIndicatorMapRec flags Field

Value	Effect
XkbIM_NoExplicit (1L<<7)	Client applications cannot change the state of the indicator.
XkbIM_NoAutomatic (1L<<6)	Xkb does not automatically change the value of the indicator based upon a change in the keyboard state, regardless of the values for the other fields of the indicator map.
XkbIM_LEDDrivesKB (1L<<5)	A client application changing the state of the indicator causes the state of the keyboard to change.

Note that if *XkbIM_NoAutomatic* is not set, by default the indicator follows the keyboard state.

If `XkbIM_LEDDrivesKB` is set and `XkbIM_NoExplicit` is not, and if you call a function which updates the server's image of the indicator map (such as `XkbSetIndicatorMap` or `XkbSetNamedIndicator`), Xkb changes the keyboard state and controls to reflect the other fields of the indicator map, as described in the remainder of this section. If you attempt to explicitly change the value of an indicator for which `XkbIM_LEDDrivesKB` is absent or for which `XkbIM_NoExplicit` is present, keyboard state or controls are unaffected.

For example, a keyboard designer may want to make the CapsLock LED controllable only by the server, but allow the Scroll Lock LED to be controlled by client applications. To do so, the keyboard designer could set the `XkbIM_NoExplicit` flag for the CapsLock LED, but not set it for the Scroll Lock LED. Or, the keyboard designer may wish to allow the CapsLock LED to be controlled by both the server and client applications, and also have the server to automatically change the CapsLock modifier state whenever a client application changes the CapsLock LED. To do so, the keyboard designer would not set the `XkbIM_NoExplicit` flag, but would instead set the `XkbIM_LEDDrivesKB` flag.

The remaining fields in the indicator map specify the conditions under which Xkb automatically turns an indicator on or off (only if `XkbIM_NoAutomatic` is not set). If these conditions match the keyboard state, Xkb turns the indicator on. If the conditions do not match, Xkb turns the indicator off.

XkbIndicatorMapRec which_groups and groups fields

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:

```
#define XkbIM_UseNone          0
#define XkbIM_UseBase          (1L << 0)
#define XkbIM_UseLatched        (1L << 1)
#define XkbIM_UseLocked          (1L << 2)
#define XkbIM_UseEffective        (1L << 3)
#define XkbIM_UseAnyGroup      XkbIM_UseLatched | XkbIM_UseLocked |
                                XkbIM_UseEffective
```

The `groups` field specifies what keyboard groups an indicator watches and is the bitwise inclusive OR of the following valid values:

```
#define XkbGroup1Mask        (1<<0)
#define XkbGroup2Mask        (1<<1)
#define XkbGroup3Mask        (1<<2)
#define XkbGroup4Mask        (1<<3)
#define XkbAnyGroupMask      (1<<7)
#define XkbAllGroupsMask     (0xf)
```

If `XkbIM_NoAutomatic` is not set (the keyboard drives the indicator), the effect of `which_groups` and `groups` is shown in Table 8.2.

Table 8.2 XkbIndicatorMapRec which_groups and groups, Keyboard Drives Indicator

<code>which_groups</code>	<code>Effect</code>
<code>XkbIM_UseNone</code>	The <code>groups</code> field and the current keyboard group state are ignored.
<code>XkbIM_UseBase</code>	If <code>groups</code> is non-zero, the indicator is lit whenever the base keyboard group is non-zero. If <code>groups</code> is zero, the indicator is lit whenever the base keyboard group is zero.
<code>XkbIM_UseLatched</code>	If <code>groups</code> is non-zero, the indicator is lit whenever the latched keyboard group is non-zero. If <code>groups</code> is zero, the indicator is lit whenever the latched keyboard group is zero.
<code>XkbIM_UseLocked</code>	The <code>groups</code> 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 <code>groups</code> .
<code>XkbIM_UseEffective</code>	The <code>groups</code> 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 <code>groups</code> .

The effect of `which_groups` and `groups` when you change an indicator for which `XkbIM_LEDDrivesKB` is set (the indicator drives the keyboard) is shown in Table 8.3. The “New State” column refers to the new state to which you set the indicator.

Table 8.3 XkbIndicatorMapRec which_groups and groups, Indicator Drives Keyboard

<code>which_groups</code>	<code>New State</code>	<code>Effect on Keyboard Group State</code>
<code>XkbIM_UseNone</code>	On or Off	No effect
<code>XkbIM_UseBase</code>	On or Off	No effect
<code>XkbIM_UseLatched</code>	On	The <code>groups</code> field is treated as a group mask. The keyboard group latch is changed to the lowest numbered group specified in <code>groups</code> ; if <code>groups</code> is empty, the keyboard group latch is changed to zero.
<code>XkbIM_UseLatched</code>	Off	The <code>groups</code> 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 <code>groups</code> ; if <code>groups</code> is zero, the keyboard group latch is set to the index of the highest legal keyboard group.
<code>XkbIM_UseLocked</code> or <code>XkbIM_UseEffective</code>	On	If the <code>groups</code> mask is empty, group is not changed, otherwise the locked keyboard group is changed to the lowest numbered group specified in <code>groups</code> .
<code>XkbIM_UseLocked</code> or <code>XkbIM_UseEffective</code>	Off	Locked keyboard group is changed to the lowest numbered group that is not specified in the <code>groups</code> mask, or to Group1 if the <code>groups</code> mask contains all keyboard groups.

XkbIndicatorMapRec which_mods and mods fields

The `mods` field specifies what modifiers an indicator watches. The `mods` field is an Xkb modifier definition, `XkbModsRec`, as described in section 7.2, 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. To specify the `mods` field, in general, assign the modifiers of interest to `mods.real_mods` and the virtual modifiers of interest to `mods.vmods`. You can disregard the `mods.mask` field unless your application needs to interpret the indicator map directly (that is, to simulate automatic indicator behavior on its own). Rela-

tively few applications need to do so, but if you find it necessary, you can either read the indicator map back from the server after you update it (the server automatically updates the mask field whenever any of the real or virtual modifiers are changed in the modifier definition) or you can call *XkbVirtualModsToReal* to determine the proper contents for the mask field, assuming that the *XkbDescRec* contains the virtual modifier definitions.

which_mods specifies what criteria Xkb uses to determine a match with the corresponding *mods* field by specifying one or more components of the Xkb keyboard state. If *XkbIM_NoAutomatic* is not set (the keyboard drives the indicator), the indicator is lit whenever any of the modifiers specified in the *mask* field of the *mods* modifier definition are also set in any of the current keyboard state components specified by *which_mods*. Remember that the *mask* field is comprised of all of the real modifiers specified in the definition plus any real modifiers that are bound to the virtual modifiers specified in the definition. (See Chapter 5 for more information on the keyboard state and Chapter 7 for more information on virtual modifiers.) Use a bitwise inclusive OR of the following values to compose a value for *which_mods*:

```
#define XkbIM_UseNone      0
#define XkbIM_UseBase      (1L << 0)
#define XkbIM_UseLatched   (1L << 1)
#define XkbIM_UseLocked    (1L << 2)
#define XkbIM_UseEffective (1L << 3)
#define XkbIM_UseCompat    (1L << 4)
#define XkbIM_UseAnyMods   XkbIM_UseBase | XkbIM_UseLatched | XkbIM_UseLocked
                           | XkbIM_UseEffective | XkbIM_UseCompat
```

If *XkbIM_NoAutomatic* is not set (the keyboard drives the indicator), the effect of *which_mods* and *mods* is shown in Table 8.4

Table 8.4 XkbIndicatorMapRec which_mods and mods, Keyboard Drives Indicator

<i>which_mods</i>	<i>Effect on Keyboard Modifiers</i>
XkbIM_UseNone	The mods field and the current keyboard modifier state are ignored
XkbIM_UseBase	The indicator is lit when any of the modifiers specified in the <i>mask</i> field of <i>mods</i> are on in the keyboard base state. If both <i>mods.real_mods</i> and <i>mods.vmods</i> are zero, the indicator is lit when the base keyboard state contains no modifiers.
XkbIM_UseLatched	The indicator is lit when any of the modifiers specified in the <i>mask</i> field of <i>mods</i> are latched. If both <i>mods.real_mods</i> and <i>mods.vmods</i> are zero, the indicator is lit when none of the modifier keys are latched.
XkbIM_UseLocked	The indicator is lit when any of the modifiers specified in the <i>mask</i> field of <i>mods</i> are locked. If both <i>mods.real_mods</i> and <i>mods.vmods</i> are zero, the indicator is lit when none of the modifier keys are locked.
XkbIM_UseEffective	The indicator is lit when any of the modifiers specified in the <i>mask</i> field of <i>mods</i> are in the effective keyboard state. If both <i>mods.real_mods</i> and <i>mods.vmods</i> are zero, the indicator is lit when the effective keyboard state contains no modifiers.
XkbIM_UseCompat	The indicator is lit when any of the modifiers specified in the <i>mask</i> field of <i>mods</i> are in the keyboard compatibility state. If both <i>mods.real_mods</i> and <i>mods.vmods</i> are zero, the indicator is lit when the keyboard compatibility state contains no modifiers.

The effect on the keyboard modifiers of *which_mods* and *mods* when you change an indicator for which `XkbIM_LEDDrivesKB` is set (the indicator drives the keyboard) is shown in Table 8.5. The “New State” column refers to the new state to which you set the indicator.

Table 8.5 XkbIndicatorMapRec *which_mods* and *mods*, Indicator Drives Keyboard

<i>which_mods</i>	<i>New State</i>	<i>Effect on Keyboard Modifiers</i>
<code>XkbIM_UseNone</code> or <code>XkbIM_UseBase</code>	On or Off	No Effect
<code>XkbIM_UseLatched</code>	On	Any modifiers specified in the <i>mask</i> field of <i>mods</i> are added to the latched modifiers.
<code>XkbIM_UseLatched</code>	Off	Any modifiers specified in the <i>mask</i> field of <i>mods</i> are removed from the latched modifiers.
<code>XkbIM_UseLocked</code> , <code>XkbIM_UseCompat</code> , or <code>XkbIM_UseEffective</code>	On	Any modifiers specified in the <i>mask</i> field of <i>mods</i> are added to the locked modifiers.
<code>XkbIM_UseLocked</code>	Off	Any modifiers specified in the <i>mask</i> field of <i>mods</i> are removed from the locked modifiers.
<code>XkbIM_UseCompat</code> or <code>XkbIM_UseEffective</code>	Off	Any modifiers specified in the <i>mask</i> field of <i>mods</i> are removed from both the locked and latched modifiers.

XkbIndicatorMapRec *ctrls* field

The *ctrls* field specifies what controls (see Chapter 10) the indicator watches and is composed using the bitwise inclusive OR of the following values:

```
#define XkbRepeatKeysMask      (1L << 0)
#define XkbSlowKeysMask        (1L << 1)
#define XkbBounceKeysMask       (1L << 2)
#define XkbStickyKeysMask       (1L << 3)
#define XkbMouseKeysMask        (1L << 4)
#define XkbMouseKeysAccelMask   (1L << 5)
#define XkbAccessXKeysMask      (1L << 6)
#define XkbAccessXTimeoutMask   (1L << 7)
#define XkbAccessXFeedbackMask  (1L << 8)
#define XkbAudibleBellMask      (1L << 9)
#define XkbOverlay1Mask         (1L << 10)
#define XkbOverlay2Mask         (1L << 11)
#define XkbAllBooleanCtrlsMask  (0x00001FFF)
```

Xkb lights the indicator whenever any of the boolean controls specified in *ctrls* is enabled.

8.3 Getting Information About Indicators

Xkb allows applications to obtain information about indicators using two different methods. The first method, which is similar to the core X implementation, uses a mask to specify the indicators. The second method, which is more suitable for applications concerned with inter-operability, uses indicator names. The correspondence between the indicator name and the bit position in masks is as follows: one of the parameters returned from `XkbGetNamedIndicators` is an index which is the bit position to use in any function call which

requires a mask of indicator bits, as well as the indicator’s index into the `XkbIndicatorRec` array of indicator maps.

8.3.1 Getting Indicator State

Because the state of the indicators are relatively volatile, the keyboard description does not hold the current state of the indicators. To obtain the current state of the keyboard indicators, call `XkbGetIndicatorState`.

```
Status XkbGetIndicatorState(display, device_spec, state_return)
    Display *      display;      /* connection to the X server */
    unsigned int   device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int * state_return; /* backfilled with a mask of the indicator state */
```

`XkbGetIndicatorState` queries the *display* for the state of the indicators on the device specified by the *device_spec*. For each indicator that is “turned on” on the device, the associated bit is set in *state_return*. If a compatible version of the Xkb extension is not available in the server `XkbGetIndicatorState` returns a `BadMatch` error. Otherwise, it sends the request to the X server, places the state of the indicators into *state_return*, and returns `Success`. Thus the value reported by `XkbGetIndicatorState` is identical to the value reported by the core protocol.

8.3.2 Getting Indicator Information by Index

To get the map for one or more indicators, using a mask to specify the indicators, call `XkbGetIndicatorMap`.

```
Status XkbGetIndicatorMap(dpy, which, desc)
    Display *      dpy;      /* connection to the X server */
    unsigned int   which;   /* mask of indicators for which maps should be returned */
    XkbDescPtr   desc;   /* keyboard description to be updated */
```

`XkbGetIndicatorMap` obtains the maps from the server for only those indicators specified by the *which* mask, and copies the values into the keyboard description specified by *desc*. If the *indicators* field of the *desc* parameter is `NULL`, `XkbGetIndicatorMap` allocates and initializes it.

`XkbGetIndicatorMap` can generate `BadAlloc`, `BadLength`, `BadMatch`, and `BadImplementation` errors.

To free the indicator maps, use `XkbFreeIndicatorMaps` (see section 8.6).

8.3.3 Getting Indicator Information by Name

Xkb also allows applications to refer to indicators by name. Call `XkbGetNames` to get the indicator names (see Chapter 18). Using names eliminates the need for hard-coding bit-mask values for particular keyboards. For example, instead of using vendor specific constants like `WSKBLED_ScrollLock` mask on Digital workstations or `XLED_SCROLL_LOCK` on Sun workstations, you can instead call `XkbGetNamedIndicator` to lookup information on the indicator named “Scroll Lock.”

Call *XkbGetNamedIndicator* to look up the indicator map and other information for an indicator by name.

```
Bool XkbGetNamedIndicator(dpy, dev_spec, name, ndx_rtrn, state_rtrn, map_rtrn, real_rtrn)
    Display *      dpy;          /* connection to the X server */
    unsigned int   device_spec; /* keyboard device id, or XkbUseCoreKbd */
    Atom          name;        /* name of the indicator to be retrieved */
    int *         ndx_rtrn;    /* backfilled with the index of the retrieved indicator */
    Bool *        state_rtrn;   /* backfilled with the current state of the retrieved indicator */
    XkbIndicatorMapPtr map_rtrn; /* backfilled with the mapping for the retrieved indicator */
    Bool *        real_rtrn;    /* backfilled with True if the named indicator is real (physical) */
```

If the device specified by *device_spec* has an indicator named *name*, *XkbGetNamedIndicator* returns True and populates the rest of the parameters with information about the indicator. Otherwise, *XkbGetNamedIndicator* returns False.

The *ndx_rtrn* field returns the 0-based index of the named indicator. This index is the bit position to use in any function call which requires a mask of indicator bits, as well as the indicator's index into the *XkbIndicatorRec* array of indicator maps. *state_rtrn* returns the current state of the named indicator (True = on, False = off). *map_rtrn* returns the indicator map for the named indicator. In addition, if the indicator is mapped to a physical LED, the *real_rtrn* parameter is set to True.

Each of the “_rtrn” arguments is optional; you can pass NULL for any unneeded “_rtrn” arguments.

XkbGetNamedIndicator can generate BadAtom and BadImplementation errors.

8.4 Changing Indicator Maps and State

Just as you can get the indicator map using a mask or using an indicator name, so you can change it using a mask or a name.

Note You cannot change the *phys_indicators* field of the indicators structure. The only way to change the *phys_indicators* field is to change the keyboard map.

There are two ways to make changes to indicator maps and state: either change a local copy of the indicator maps and call *XkbSetIndicatorMap* or *XkbSetNamedIndicator*, or, to reduce network traffic, use an *XkbIndicatorChangesRec* structure and call *XkbChangeIndicators*.

8.4.1 Effects of Explicit Changes on Indicators

This section discusses the effects of explicitly changing indicators depending upon different settings in the indicator map. See Table 8.3 on page 38 and Table 8.5 on page 40 for information on the effects of the indicator map fields when explicit changes are made.

If *XkbIM_LEDDrivesKB* is set and *XkbIM_NoExplicit* is not, and if you call a function which updates the server's image of the indicator map (such as *XkbSetIndicatorMap* or *XkbSetNamedIndicator*), Xkb changes the keyboard state and controls to reflect the other fields of the indicator map. If you attempt to explicitly change the value of an indicator for which *XkbIM_LEDDrivesKB* is absent or for which *XkbIM_NoExplicit* is present, keyboard state or controls are unaffected.

If neither `XkbIM_NoAutomatic` nor `XkbIM_NoExplicit` is set in an indicator map, Xkb honors any request to change the state of the indicator, but the new state might be immediately superseded by automatic changes to the indicator state if the keyboard state or controls change.

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 you change an indicator for which both the `XkbIM_LEDDrivesKB` and `XkbIM_NoAutomatic` flags are specified, Xkb applies the keyboard changes specified in the other indicator map fields and changes the indicator to reflect the state that was explicitly requested. The indicator remains in the new state until it is explicitly changed again.

If the `XkbIM_NoAutomatic` flag is not set and `XkbIM_LEDDrivesKB` is set, Xkb applies the changes specified in the other indicator map fields and sets the state of the indicator 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, Xkb does not extinguish an indicator with `which_mods` of `XkbIM_UseBase` and `mods` of `Shift` if, at the time Xkb processes the request to extinguish the indicator, one of the Shift keys is physically depressed.

If you explicitly light an indicator for which `XkbIM_LEDDrivesKB` is set, Xkb enables all of the boolean controls specified in the `ctrls` field of its indicator map. Explicitly extinguishing such an indicator causes Xkb to disable all of the boolean controls specified in `ctrls`.

8.4.2 Changing Indicator Maps by Index

To update the maps for one or more indicators, first modify a local copy of the keyboard description, then use `XkbSetIndicatorMap` to download the changes to the server:

```
Bool XkbSetIndicatorMap(dpy, which, desc)
    Display * dpy; /* connection to the X server */
    unsigned int which; /* mask of indicators to change */
    XkbDescPtr desc; /* keyboard description from which the maps are taken */
```

For each bit set in the `which` parameter, `XkbSetIndicatorMap` sends the corresponding indicator map from the `desc` parameter to the server.

8.4.3 Changing Indicator Maps by Name

XkbSetNamedIndicator can do several related things:

- Name an indicator if it is not already named
- Toggle the state of the indicator
- Set the indicator to a specified state
- Set the indicator map for the indicator

```
Bool XkbSetNamedIndicator(dpy, device_spec, name, change_state, state, create_new, map)
Display *          dpy;           /* connection to the X server */
unsigned int       device_spec; /* device id, or XkbUseCoreKbd */
Atom              name;          /* name of the indicator to change */
Bool              change_state; /* whether to change the indicator state or not */
Bool              state;          /* desired new state for the indicator */
Bool              create_new;    /* whether a new indicator with the specified name
                                should be created when necessary */
XkbIndicatorMapPtr map;         /* new map for the indicator */
```

If a compatible version of the Xkb extension is not available in the server, *XkbSetNamedIndicator* returns `False`. Otherwise, it sends a request to the X server to change the indicator specified by *name* and returns `True`.

If *change_state* is `True`, and the optional parameter, *state*, is not `NULL`, *XkbSetNamedIndicator* tells the server to change the state of the named indicator to the value specified by *state*.

If an indicator with the name specified by *name* does not already exist, the *create_new* parameter tells the server whether it should create a new named indicator. If *create_new* is `True`, the server finds the first indicator that doesn't have a name and gives it the name specified by *name*.

If the optional parameter, *map*, is not `NULL`, *XkbSetNamedIndicator* tells the server to change the indicator's map to the values specified in *map*.

XkbSetNamedIndicator can generate `BadAtom` and `BadImplementation` errors. In addition, it can also generate `XkbIndicatorStateNotify` (see section 8.5), `XkbIndicatorMapNotify`, and `XkbNamesNotify` events (see section 18.5).

8.4.4 The `XkbIndicatorChangesRec` Structure

The `XkbIndicatorChangesRec` identifies small modifications to the indicator map. Use it with the function *XkbChangeIndicators* to reduce the amount of traffic sent to the server.

```
typedef struct _XkbIndicatorChanges {
    unsigned int      state_changes;
    unsigned int      map_changes;
} XkbIndicatorChangesRec, *XkbIndicatorChangesPtr;
```

The *state_changes* field is a mask that specifies the indicators that have changed state, and *map_changes* is a mask that specifies the indicators whose maps have changed.

To change indicator maps or state without passing the entire keyboard description, call *XkbChangeIndicators*.

```
Bool XkbChangeIndicators(dpy, xkb, changes, state)
    Display *           dpy;      /* connection to the X server */
    XkbDescPtr         xkb;      /* keyboard description from which names are to be
                                    taken. */
    XkbIndicatorChangesPtr changes; /* indicators to be updated on the server */
    unsigned int        state;    /* new state of indicators listed in
                                    changes->state_changes */
```

XkbChangeIndicators copies any maps specified by *changes* from the keyboard description, *xkb*, to the server specified by *dpy*. If any bits are set in the *state_changes* field of *changes*, *XkbChangeIndicators* also sets the state of those indicators to the values specified in the *state* mask. A 1 bit in *state* turns the corresponding indicator on, a 0 bit turns it off.

XkbChangeIndicators can generate `BadAtom` and `BadImplementation` errors. In addition, it can also generate `XkbIndicatorStateNotify` and `XkbIndicatorMapNotify` events (see section 8.5).

8.5 Tracking Changes To Indicator State or Map

Whenever an indicator changes state, the server sends `XkbIndicatorStateNotify` events to all interested clients. Similarly, whenever an indicator's map changes, the server sends `XkbIndicatorMapNotify` events to all interested clients.

To receive `XkbIndicatorStateNotify` events, call *XkbSelectEvents* (see section 4.3) with both the *bits_to_change* and *values_for_bits* parameters containing `XkbIndicatorStateNotifyMask`. To receive `XkbIndicatorMapNotify` events, call *XkbSelectEvents* with `XkbIndicatorMapNotifyMask`.

To receive events for only specific indicators, use *XkbSelectEventDetails*. Set the *event_type* parameter to `XkbIndicatorStateNotify` or `XkbIndicatorMapNotify`, and set both the *bits_to_change* and *values_for_bits* detail parameters to a mask where each bit specifies one indicator, turning on those bits which specify the indicators for which you want to receive events.

Both types of indicator events use the same structure:

```
typedef struct _XkbIndicatorNotify {
    int          type;      /* Xkb extension base event code */
    unsigned long serial;   /* X server serial number for event */
    Bool         send_event; /* True => synthetically generated */
    Display *    display;   /* server connection where event generated */
    Time         time;     /* server time when event generated */
    int          xkb_type;  /* specifies state or map notify */
    int          device;    /* Xkb device id, will not be XkbUseCoreKbd*/
    unsigned int  changed;  /* mask of indicators with new state or map */
    unsigned int  state;    /* current state of all indicators */
} XkbIndicatorNotifyEvent;
```

xkb_type is either `XkbIndicatorStateNotify` or `XkbIndicatorMapNotify`, depending upon whether the event is a `kbIndicatorStateNotify` event or `kbIndicatorMapNotify` event.

The *changed* parameter is a mask that is the bitwise inclusive OR of the indicators that have changed. If the event is of type `XkbIndicatorMapNotify`, *changed* reports the maps that changed. If the event is of type `XkbIndicatorStateNotify`, *changed* reports the indicators that have changed state. *state* is a mask which specifies the current state of all indicators, whether they have changed or not, for both `XkbIndicatorStateNotify` and `IndicatorMapNotify` events.

When your client application receives either a `XkbIndicatorStateNotify` event or `XkbIndicatorMapNotify` event, you can note the changes in a *changes* structure by calling `XkbNoteIndicatorChanges`.

```
void XkbNoteIndicatorChanges(old, new, wanted)
    XkbIndicatorChangesPtr old;      /* XkbIndicatorChanges structure to be updated */
    XkbIndicatorNotifyEvent * new;      /* event from which changes are to be copied */
    unsigned int           wanted;     /* which changes are to be noted */
```

The *wanted* parameter is the bitwise inclusive OR of `XkbIndicatorMapMask` and `XkbIndicatorStateMask`. `XkbNoteIndicatorChanges` copies any changes reported in *new* and specified in *wanted* into the *changes* record specified by *old*.

To update a local copy of the keyboard description with the actual values, pass the results of one or more calls to `XkbNoteIndicatorChanges` to `XkbGetIndicatorChanges`:

```
Status XkbGetIndicatorChanges(dpy, xkb, changes, state)
    Display *          dpy;        /* connection to the X server */
    XkbDescPtr        xkb;        /* keyboard description to hold the new values */
    XkbIndicatorChangesPtr changes; /* indicator maps/state to be obtained from the server */
    unsigned int *    state;       /* backfilled with the state of the indicators */
```

`XkbGetIndicatorChanges` examines the *changes* parameter, pulls over the necessary information from the server, and copies the results into the *xkb* keyboard description. If any bits are set in the *state_changes* field of *changes*, `XkbGetIndicatorChanges` also places the state of those indicators in *state*. If the *indicators* field of *xkb* is `NULL`, `XkbGetIndicatorChanges` allocates and initializes it. To free the *indicators* field, use `XkbFreeIndicators` (see section 8.6)

`XkbGetIndicatorChanges` can generate `BadAlloc`, `BadImplementation` and `BadMatch` errors.

8.6 Allocating and Freeing Indicator Maps

Most applications do not need to directly allocate the *indicators* member of the keyboard description record (the keyboard description record is described in Chapter 6). If the need arises, however, call `XkbAllocIndicatorMaps`.

```
Status XkbAllocIndicatorMaps(xkb)
    XkbDescPtr xkb;      /* keyboard description structure */
```

The *xkb* parameter must point to a valid keyboard description. If it doesn't, `XkbAllocIndicatorMaps` returns a `BadMatch` error. Otherwise, `XkbAllocIndicatorMaps` allocates and initializes the *indicators* member of the keyboard description record and returns `Suc-`

cess. If *XkbAllocIndicatorMaps* was unable to allocate the indicators record, it reports a BadAlloc error.

To free memory used by the *indicators* member of an *XkbDescRec* structure, call *XkbFreeIndicatorMaps*.

```
void XkbFreeIndicatorMaps(xkb)
    XkbDescPtr xkb;      /* keyboard description structure */
```

If the *indicators* member of the keyboard description record pointed to by *xkb* is not NULL, *XkbFreeIndicatorMaps* frees the memory associated with the *indicators* member of *xkb*.

9 Bells

The core X protocol only allows applications to explicitly sound the system bell with a given duration, pitch, and volume. Xkb extends this capability by allowing clients to attach symbolic names to bells, disable audible bells, and receive an event whenever the keyboard bell is rung. For the purposes of this document, the *audible* bell is defined to be the system bell, or the default keyboard bell, as opposed to any other audible sound generated elsewhere in the system.

You can ask to receive `XkbBellNotify` events (see section 9.4) when any client rings any one of the following:

- the default bell
- any bell on an input device that can be specified by a *bell_class* and *bell_id* pair
- any bell specified only by an arbitrary name (This is, from the server's point of view, merely a name, and not connected with any physical sound-generating device. Some client application must generate the sound, or visual feedback, if any, that is associated with the name.)

You can also ask to receive `XkbBellNotify` events when the server rings the default bell or if any client has requested events only (without the bell sounding) for any of the bell types shown above.

You can disable audible bells on a global basis (to set the `AudibleBell` control, see Chapter 10). For example, a client that replaces the keyboard bell with some other audible cue might want to turn off the `AudibleBell` control to prevent the server from also generating a sound and avoid cacophony. If you disable audible bells and request to receive `XkbBellNotify` events, you can generate feedback different from the default bell.

You can, however, override the `AudibleBell` control, by calling one of the functions that force the ringing of a bell in spite of the setting of the `AudibleBell` control — `XkbForceDeviceBell` or `XkbForceBell` (see section 9.3.3). In this case the server does not generate a bell event.

Just as some keyboards can produce keyclicks to indicate when a key is pressed or repeating, Xkb can provide feedback for the controls by using special beep codes. The `AccessXFeedback` control is used to configure the specific types of operations which generate feedback. See section 10.6.3 on page 64 for a discussion on `AccessXFeedback` control.

This chapter describes bell names, the routines used to generate named bells, and the events the server generates for bells.

9.1 Bell Names

You can associate a name to an act of ringing a bell by converting the name to an Atom and then using this name when you call the functions listed in this chapter. If an event is generated as a result, the name is then passed to all other clients interested in receiving `XkbBellNotify` events. Note that these are arbitrary names and there is no binding to any sounds. Any sounds or other effects (such as visual bells on the screen) must be generated by a client application upon receipt of the bell event containing the name. There is no default name for the default keyboard bell. The server does generate some predefined bells for the AccessX controls (see section 10.6.3). These named bells are shown in Table 9.1;

the name is included in any bell event sent to clients that have requested to receive `XkbBellNotify` events.

Table 9.1 Predefined Bells

Action	Named Bell
Indicator turned on	<code>AX_IndicatorOn</code>
Indicator turned off	<code>AX_IndicatorOff</code>
More than one indicator changed state	<code>AX_IndicatorChange</code>
Control turned on	<code>AX_FeatureOn</code>
Control turned off	<code>AX_FeatureOff</code>
More than one control changed state	<code>AX_FeatureChange</code>
SlowKeys and BounceKeys about to be turned on or off	<code>AX_SlowKeysWarning</code>
SlowKeys key pressed	<code>AX_SlowKeyPress</code>
SlowKeys key accepted	<code>AX_SlowKeyAccept</code>
SlowKeys key rejected	<code>AX_SlowKeyReject</code>
Accepted SlowKeys key released	<code>AX_SlowKeyRelease</code>
BounceKeys key rejected	<code>AX_BounceKeyReject</code>
StickyKeys key latched	<code>AX_StickyLatch</code>
StickyKeys key locked	<code>AX_StickyLock</code>
StickyKeys key unlocked	<code>AX_StickyUnlock</code>

9.2 Audible Bells

Using Xkb you can generate bell events which do not necessarily ring the system bell. This is useful if you need to use an audio server instead of the system beep. For example, when an audio client starts, it could disable the audible bell (the system bell) and then listen for `XkbBellNotify` events (see section 9.4). When it receives a `XkbBellNotify` event, the audio client could then send a request to an audio server to play a sound.

You can control the audible bells feature by passing the `XkbAudibleBellMask` to `XkbChangeEnabledControls` (see section 10.1.1). If you set `XkbAudibleBellMask` on, the server rings the system bell when a bell event occurs. This is the default. If you set `XkbAudibleBellMask` off and a bell event occurs, the server does not ring the system bell unless you call `XkbForceDeviceBell` or `XkbForceBell` (see section 9.3.3).

Audible bells are also part of the per-client auto-reset controls. For more information on auto-reset controls, see section 10.1.2 on page 55.

9.3 Bell Functions

Use the functions described in this section to ring bells and to generate bell events.

The input extension has two types of feedbacks that can generate bells — bell feedback and keyboard feedback. Some of the functions in this section have `bell_class` and `bell_id` parameters; set them as follows: Set `bell_class` to `BellFeedbackClass` or `KbdFeedbackClass`. A device can have more than one feedback of each type; set `bell_id` to the particular bell feedback of `bell_class` type.

Table 9.2 shows the conditions that cause a bell to sound or an `XkbBellNotifyEvent` to be generated when a bell function is called.

Table 9.2 Bell Sounding and Bell Event Generating

<i>Function Called</i>	<i>AudibleBell</i>	<i>Server sounds a bell</i>	<i>Server sends an XkbBellNotifyEvent</i>
<code>XkbDeviceBell</code>	On	Yes	Yes
<code>XkbDeviceBell</code>	Off	No	Yes
<code>XkbBell</code>	On	Yes	Yes
<code>XkbBell</code>	Off	No	Yes
<code>XkbDeviceBellEvent</code>	On or Off	No	Yes
<code>XkbBellEvent</code>	On or Off	No	Yes
<code>XkbDeviceForceBell</code>	On or Off	Yes	No
<code>XkbForceBell</code>	On or Off	Yes	No

9.3.1 Generating Named Bells

To ring the bell on an X input extension device or the default keyboard, call `XkbDeviceBell`.

```
Bool XkbDeviceBell(display, window, device_id, bell_class, bell_id, percent, name)
    Display * display;      /* connection to the X server */
    Window   window;       /* window for which the bell is generated, or None */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int bell_class; /* X input extension bell class of the bell to be rung */
    unsigned int bell_id;   /* X input extension bell id of the bell to be rung */
    int        percent;     /* bell volume, from -100 to 100 inclusive */
    Atom      name;        /* a name for the bell, or NULL */
```

Set `percent` to be the volume relative to the base volume for the keyboard as described for `XBell`.

Note that `bell_class` and `bell_id` indicate the bell to physically ring. `name` is simply an arbitrary moniker for the client application's use.

To determine the current feedback settings of an extension input device, call `XGetFeedbackControl`. See the X input extension documentation for more information on `XGetFeedbackControl` and related data structures.

If a compatible keyboard extension is not present in the X server, `XkbDeviceBell` immediately returns False. Otherwise, `XkbDeviceBell` rings the bell as specified for the display and keyboard device, and returns True. If you have disabled the audible bell, the server does not ring the system bell, although it does generate a `XkbBellNotify` event.

You can call `XkbDeviceBell` without first initializing the keyboard extension.

As a convenience routine, Xkb provides a function to ring the bell on the default keyboard: `XkbBell`.

```
Bool XkbBell(display, window, percent, name)
    Display * display;      /* connection to the X server */
    Window   window;       /* event window, or None */
    int        percent;     /* relative volume, which can range from -100 to 100 inclusive */
    Atom      name;        /* a bell name, or NULL */
```

If a compatible keyboard extension isn't present in the X server, *XkbBell* calls *XBell* with the specified *display* and *percent*, and returns False. Otherwise, *XkbBell* calls *XkbDeviceBell* with the specified *display*, *window*, *percent*, and *name*, a *device_spec* of *XkbUseCoreKbd*, a *bell_class* of *XkbDfltXIClass*, and a *bell_id* of *XkbDfltXIID*, and returns True.

If you have disabled the audible bell, the server does not ring the system bell, although it does generate a *XkbBellNotify* event.

You can call *XkbBell* without first initializing the keyboard extension.

9.3.2 Generating Named Bell Events

Using Xkb, you can also generate a named bell event that does not ring any bell. This allows you to do things such as generate events when your application starts.

For example, if an audio client listens for these types of bells, it can produce a “whoosh” sound when it receives a named bell event to indicate a client just started. In this manner, applications can generate start-up feedback and not worry about producing annoying beeps if an audio server is not running.

To cause a bell event for an X input extension device or for the keyboard, without ringing the corresponding bell, call *XkbDeviceBellEvent*.

```
Bool XkbDeviceBellEvent(display, window, device_spec, bell_class, bell_id, percent, name)
    Display *    display; /* connection to the X server */
    Window       window; /* event window, or None */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int bell_class; /* input extension bell class for the event */
    unsigned int bell_id;   /* input extension bell id for the event */
    int          percent;  /* volume for the bell, which can range from -100 to 100 inclusive */
    Atom         name;    /* a bell name, or NULL */
```

If a compatible keyboard extension isn't present in the X server, *XkbDeviceBellEvent* immediately returns False. Otherwise, *XkbDeviceBellEvent* causes an *XkbBellNotify* event to be sent to all interested clients and returns True. Set *percent* to be the volume relative to the base volume for the keyboard as described for *XBell*.

In addition, *XkbDeviceBellEvent* may generate Atom protocol errors as well as *XkbBellNotify* events. You can call *XkbBell* without first initializing the keyboard extension.

As a convenience routine, Xkb provides a function to cause a bell event for the keyboard without ringing the bell: *XkbBellEvent*.

```
Bool XkbBellEvent(display, window, percent, name)
    Display *    display; /* connection to the X server */
    Window       window; /* the event window, or None */
    int          percent; /* relative volume, which can range from -100 to 100 inclusive */
    Atom         name;  /* a bell name, or NULL */
```

If a compatible keyboard extension isn't present in the X server, *XkbBellEvent* immediately returns False. Otherwise, *XkbBellEvent* calls *XkbDeviceBellEvent* with the specified *display*, *window*, *percent*, and *name*, a *device_spec* of *XkbUseCoreKbd*, a *bell_class*

of `XkbDfltXIClass`, and a `bell_id` of `XkbDfltXIId`, and returns what `XkbDeviceBellEvent` returns.

`XkbBellEvent` generates a `XkbBellNotify` event.

You can call `XkbBellEvent` without first initializing the keyboard extension.

9.3.3 Forcing a Server-generated Bell

To ring the bell on any keyboard, overriding user preference settings for audible bells, call `XkbForceDeviceBell`.

```
Bool XkbForceDeviceBell(display, window, device_spec, bell_class, bell_id, percent)
    Display * display;      /* connection to the X server */
    Window   window;       /* event window, or None */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int bell_class; /* input extension class of the bell to be rung */
    unsigned int bell_id;   /* input extension id of the bell to be rung */
    int        percent;    /* relative volume, which can range from -100 to 100 inclusive */
```

If a compatible keyboard extension isn't present in the X server, `XkbForceDeviceBell` immediately returns `False`. Otherwise, `XkbForceDeviceBell` rings the bell as specified for the display and keyboard device, and returns `True`. Set `percent` to be the volume relative to the base volume for the keyboard as described for `XBell`. There is no `name` parameter because `XkbForceDeviceBell` does not cause an `XkbBellNotify` event.

You can call `XkbBell` without first initializing the keyboard extension.

To ring the bell on the default keyboard, overriding user preference settings for audible bells call `XkbForceBell`.

```
Bool XkbForceBell(display, percent)
    Display * display; /* connection to the X server */
    int        percent; /* volume for the bell, which can range from -100 to 100 inclusive */
```

If a compatible keyboard extension isn't present in the X server, `XkbForceBell` calls `XBell` with the specified `display` and `percent`, and returns `False`. Otherwise, `XkbForceBell` calls `XkbForceDeviceBell` with the specified `display` and `percent`, `device_spec = XkbUseCoreKbd`, `bell_class = XkbDfltXIClass`, `bell_id = XkbDfltXIId`, `window = None`, and `name = NULL`, and returns what `XkbForceDeviceBell` returns.

`XkbForceBell` does not cause an `XkbBellNotify` event.

You can call `XkbBell` without first initializing the keyboard extension.

9.4 Detecting Bells

Xkb generates `XkbBellNotify` events for all bells except for those resulting from calls to `XkbForceDeviceBell` and `XkbForceBell`. To receive `XkbBellNotify` events under all possible conditions, pass `XkbBellNotifyMask` in both the `bits_to_change` and `values_for_bits` parameters to `XkbSelectEvents` (see section 4.3).

The `XkbBellNotify` event has no event details. It is either selected or it is not. However, you can call `XkbSelectEventDetails` using `XkbBellNotify` as the `event_type` and specifying `XkbAllBellNotifyMask` in `bits_to_change` and `values_for_bits`. This has the same effect as a call to `XkbSelectEvents`.

The structure for the `XkbBellNotify` event type contains:

```
typedef struct _XkbBellNotify {  
    int          type;      /* Xkb extension base event code */  
    unsigned long serial;   /* X server serial number for event */  
    Bool         send_event; /* True => synthetically generated */  
    Display *    display;   /* server connection where event generated */  
    Time         time;     /* server time when event generated */  
    int          xkb_type;  /* XkbBellNotify */  
    unsigned int device;   /* Xkb device id, will not be XkbUseCoreKbd */  
    int          percent;   /* requested volume as % of max */  
    int          pitch;    /* requested pitch in Hz */  
    int          duration;  /* requested duration in microseconds */  
    unsigned int bell_class; /* X input extension feedback class */  
    unsigned int bell_id;   /* X input extension feedback id */  
    Atom         name;     /* "name" of requested bell */  
    Window       window;   /* window associated with event */  
    Bool         event_only; /* False -> the server did not produce a beep */  
} XkbBellNotifyEvent;
```

If your application needs to generate visual bell feedback on the screen when it receives a bell event, use the window id in the `XkbBellNotifyEvent`, if present.

10 Keyboard Controls

The Xkb extension is composed of two parts: a server extension, and a client-side X library extension. This chapter discusses functions used to modify controls effecting the behavior of the server portion of the Xkb extension. Chapter 11 discusses functions used to modify controls which effect only the behavior of the client portion of the extension; those controls are known as Library Controls.

Xkb contains control features which affect the entire keyboard, known as global keyboard controls. Some of the controls may be selectively enabled and disabled; these controls are known as the *Boolean Controls*. Boolean Controls can be turned on or off under program control and can also be automatically set to an on or off condition when a client program exits. The remaining controls, known as the *Non-Boolean Controls*, are always active. The `XkbControlsRec` structure describes the current state of most of the global controls and the attributes effecting the behavior of each of these Xkb features. This chapter describes the Xkb controls and how to manipulate them.

There are two possible components for each of the Boolean Controls: attributes describing how the control should work, and a state describing whether the behavior as a whole is enabled or disabled. The attributes and state for most of these controls are held in the `XkbControlsRec` structure (see section 10.8).

You can manipulate the Xkb controls individually, via convenience functions, or as a whole. To treat them as a group, modify an `XkbControlsRec` structure to describe all of the changes to be made, and then pass that structure and appropriate flags to an Xkb library routine, or use a `XkbControlsChangesRec` (see section 10.10.1) to reduce network traffic. When using a convenience function to manipulate one control individually, you do not use an `XkbControlsRec` structure directly.

The Xkb controls are grouped as shown in Table 10.1.

Table 10.1 Xkb Keyboard Controls

Type of Control	Control Name	Boolean Control?
Controls for enabling and disabling other controls	EnabledControls	No
	AutoReset	No
Control for bell behavior	AudibleBell	Boolean
Controls for repeat key behavior	PerKeyRepeat	No
	RepeatKeys	Boolean
	DetectableAutorepeat	Boolean
Controls for keyboard overlays	Overlay1	Boolean
	Overlay2	Boolean
Controls for using the mouse from the keyboard	MouseKeys	Boolean
	MouseKeysAccel	Boolean
Controls for better keyboard access by physically-impaired persons	AccessXFeedback	Boolean
	AccessXKeys	Boolean
	AccessXTimeout	Boolean
	BounceKeys	Boolean
	SlowKeys	Boolean
	StickyKeys	Boolean
Controls for general keyboard mapping	GroupsWrap	No

Table 10.1 Xkb Keyboard Controls

Type of Control	Control Name	Boolean Control?
	IgnoreGroupLock	Boolean
	IgnoreLockMods	No
	InternalMods	No

The individual categories and controls are described first, together with functions for manipulating them. A description of the `XkbControlsRec` structure and the general functions for dealing with all of the controls at once follows at the end of the chapter.

10.1 Controls that Enable and Disable other Controls

Enable and disable the boolean controls under program control by using the `EnabledControls` control; enable and disable them upon program exit by configuring the `AutoReset` control.

10.1.1 The EnabledControls Control

The `EnabledControls` control is a bit mask where each bit which is turned on means the corresponding control is enabled, and when turned off, disabled. It corresponds to the `enabled_ctrls` field of an `XkbControlsRec` structure (see section 10.8). The bits describing which controls are turned on or off are defined in Table 10.7 on page 74.

Use `XkbChangeEnabledControls` to manipulate the `EnabledControls` control.

Bool `XkbChangeEnabledControls(dpy, device_spec, mask, values)`

Display *	<i>dpy</i> ;	/* connection to X server */
unsigned int	<i>device_spec</i> ;	/* keyboard device to modify */
unsigned int	<i>mask</i> ;	/* 1 bit -> controls to enable / disable */
unsigned int	<i>values</i> ;	/* 1 bit => enable, 0 bit => disable */

The *mask* parameter specifies the boolean controls to be enabled or disabled, and the *values* mask specifies the new state for those controls. Valid values for both of these masks are composed of a bitwise inclusive OR of bits taken from the set of mask bits in Table 10.7 on page 74, using only those masks with “ok” in the `enabled_ctrls` column.

If the X server does not support a compatible version of Xkb or the Xkb extension has not been properly initialized, `XkbChangeEnabledControls` returns False, otherwise it sends the request to the X server and returns True.

Note that the `EnabledControls` control only enables and disables controls; it does not configure them. Some controls, such as the `AudibleBell` control, have no configuration attributes, and are therefore manipulated solely by enabling and disabling them. Others, however, have additional attributes to configure their behavior. For example, the `RepeatControl` control uses `repeat_delay` and `repeat_interval` fields to describe the timing behavior of keys which repeat. The `RepeatControl` behavior is turned on or off depending on the value of the `XkbRepeatKeysMask` bit, but you must use other means, as described in this chapter, to configure its behavior in detail.

10.1.2 The AutoReset Control

You can configure the boolean controls to automatically be enabled or disabled when a program exits. This capability is controlled via two masks maintained in the X server on a

per-client basis. There is no client-side Xkb data structure corresponding to these masks. 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. The bits used in the masks correspond to the `EnabledControls` control bits.

For example, a client that replaces the keyboard bell with some other audible cue might want to turn off the `AudibleBell` control to prevent the server from also generating a sound and 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.

To get the current values of the auto reset controls, call `XkbGetAutoResetControls`.

```
Bool XkbGetAutoResetControls(dpy, auto_ctrls, auto_values)
    Display *          dpy;           /* connection to X server */
    unsigned int *     auto_ctrls;    /* specifies which bits in auto_values are relevant */
    unsigned int *     auto_values;   /* 1 bit => corresponding control has auto-reset on */
```

`XkbGetAutoResetControls` backfills `auto_ctrls` and `auto_values` with the `AutoReset` control attributes for this particular client. It returns `True` if successful, and `False` otherwise.

To change the current values of the `AutoReset` control attributes, call `XkbSetAutoResetControls`.

```
Bool XkbSetAutoResetControls(dpy, changes, auto_ctrls, auto_values)
    Display *          dpy;           /* connection to X server */
    unsigned int        changes;       /* controls for which to change auto-reset values */
    unsigned int *     auto_ctrls;    /* controls from changes which should auto reset */
    unsigned int *     auto_values;   /* 1 bit => auto-reset on */
```

`XkbSetAutoResetControls` changes the auto-reset status and associated auto-reset values for the controls selected by `changes`. For any control selected by `changes`, if the corresponding bit is set in `auto_ctrls`, the control is configured to auto-reset when the client exits. If the corresponding bit in `auto_values` is on, the control is turned on when the client exits; if zero, the control is turned off when the client exits. For any control selected by `changes`, if the corresponding bit is not set in `auto_ctrls`, the control is configured to not reset when the client exits. For example:

To leave the auto-reset controls for `StickyKeys` the way they are:

```
ok = XkbSetAutoResetControls(dpy, 0, 0, 0);
```

To change the auto-reset controls so that `StickyKeys` are unaffected when the client exits:

```
ok = XkbSetAutoResetControls(dpy, XkbStickyKeysMask, 0, 0);
```

To change the auto-reset controls so that `StickyKeys` are turned off when the client exits:

```
ok = XkbSetAutoResetControls(dpy, XkbStickyKeysMask, XkbStickyKeysMask, 0);
```

To change the auto-reset controls so that `StickyKeys` are turned on when the client exits:

```
ok = XkbSetAutoResetControls(dpy, XkbStickyKeysMask, XkbStickyKeysMask,  
XkbStickyKeysMask);
```

XkbSetAutoResetControls backfills *auto_ctrls* and *auto_values* with the auto-reset controls for this particular client. Note that all of the bits are valid in the returned values, not just the ones selected in the *changes* mask.

10.2 Control for Bell Behavior

The X server's generation of sounds is controlled by the `AudibleBell` control. Configuration of different bell sounds is discussed in Chapter 9.

10.2.1 The AudibleBell Control

The `AudibleBell` control is a boolean control which has no attributes. As such, you may enable and disable it using either the `EnabledControls` control or the `AutoReset` control discussed in section 10.1.1. When enabled, protocol requests to generate a sound result in the X server actually producing a real sound; when disabled, requests to the server to generate a sound are ignored unless the sound is forced. See section 9.2 on page 49.

10.3 Controls for Repeat Key Behavior

The repeating behavior of keyboard keys is governed by three controls, the `PerKeyRepeat` control, which is always active, and the `RepeatKeys` and `DetectableAutorepeat` controls, which are boolean controls which may be enabled and disabled.

`PerKeyRepeat` determines which keys are allowed to repeat. `RepeatKeys` governs the behavior of an individual key when it is repeating. `DetectableAutorepeat` allows a client to detect when a key is repeating as a result of being held down.

10.3.1 The PerKeyRepeat Control

The `PerKeyRepeat` control is a bitmask long enough to contain a bit for each key on the device; it determines which individual keys are allowed to repeat. The Xkb `PerKeyRepeat` control provides no functionality different from that available via the core X protocol. There are no convenience functions in Xkb for manipulating this control. The `PerKeyRepeat` control settings are carried in the *per_key_repeat* field of an `XkbControlsRec` structure, discussed in section 10.8.

10.3.2 The RepeatKeys Control

The core protocol only allows control over whether or not the entire keyboard or individual keys should auto-repeat when held down. `RepeatKeys` is a boolean control which extends this capability by adding control over the delay until a key begins to repeat and the rate at which it repeats. `RepeatKeys` is coupled with the core auto-repeat control: when `RepeatKeys` is enabled or disabled, the core auto-repeat is enabled or disabled and vice-versa.

Auto-repeating keys are controlled by two attributes. The first, *timeout*, is the delay after the initial press of an auto-repeating key and the first generated repeat event. The second, *interval*, is the delay between all subsequent generated repeat events. As with all boolean controls, configuring the attributes which determine how the control operates does not automatically enable the control as a whole; see section 10.1.

To get the current attributes of the RepeatKeys control for a keyboard device, call *XkbGetAutoRepeatRate*.

```
Bool XkbGetAutoRepeatRate(display, device_spec, timeout_rtrn, interval_rtrn)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec;   /* desired device id, or XkbUseCoreKbd */
    unsigned int *    timeout_rtrn;  /* backfilled with initial repeat delay, ms */
    unsigned int *    interval_rtrn; /* backfilled with subsequent repeat delay, ms */
```

XkbGetAutoRepeatRate queries the server for the current values of the RepeatControls control attributes, backfills *timeout_rtrn* and *interval_rtrn* with them, and returns True. If a compatible version of the Xkb extension is not available in the server *XkbGetAutoRepeatRate* returns False.

To set the attributes of the RepeatKeys control for a keyboard device, call *XkbSetAutoRepeatRate*.

```
Bool XkbSetAutoRepeatRate(display, device_spec, timeout, interval)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec;   /* device to configure, or XkbUseCoreKbd */
    unsigned int       timeout;     /* initial delay, ms */
    unsigned int       interval;    /* delay between repeats, ms */
```

XkbSetAutoRepeatRate sends a request to the X server to configure the AutoRepeat control attributes to the values specified in *timeout* and *interval*.

XkbSetAutoRepeatRate does not wait for a reply; it normally returns True. Specifying a zero value for either *timeout* or *interval* causes the server to generate a BadValue protocol error. If a compatible version of the Xkb extension is not available in the server, *XkbSetAutoRepeatRate* returns False.

10.3.3 The DetectableAutorepeat Control

Auto-repeat is the generation of multiple key events by a keyboard when the user presses a key and holds it down. Keyboard hardware and device-dependent X server software often implement auto-repeat by generating multiple KeyPress events with no intervening KeyRelease event. The standard behavior of the X server is to generate a KeyRelease event for every KeyPress event. If the keyboard hardware and device-dependent software of the X server implement auto-repeat by generating multiple KeyPress events, the device-independent part of the X server by default synthetically generates a KeyRelease event after each KeyPress event. This provides predictable behavior for X clients, but does not allow those clients to detect the fact that a key is auto-repeating.

Xkb allows clients to request *detectable auto-repeat*. If a client requests and the server supports DetectableAutorepeat, Xkb generates KeyRelease events only when the key is physically released. If DetectableAutorepeat is not supported or has not been requested, the server synthesizes a KeyRelease event for each repeating KeyPress event it generates.

DetectableAutorepeat, unlike the other controls in this chapter, is not contained in the XkbControlsRec structure, nor can it be enabled or disabled via the EnabledControls control. Instead, query and set DetectableAutorepeat using *XkbGetDetectableAutorepeat* and *XkbSetDetectableAutorepeat*.

`DetectableAutorepeat` is a condition which applies to all keyboard devices for a client's connection to a given X server; it cannot be selectively set for some devices and not for others. For this reason, none of the Xkb library functions involving `DetectableAutorepeat` involve a device specifier.

To determine whether or not the server supports `DetectableAutorepeat`, call *XkbGetDetectableAutorepeat*.

```
Bool XkbGetDetectableAutorepeat(display, supported_rtrn)
    Display * display;          /* connection to X server */
    Bool *     supported_rtrn; /* backfilled True if DetectableAutorepeat supported */
```

XkbGetDetectableAutorepeat queries the server for the current state of `DetectableAutorepeat` and waits for a reply. If *supported_rtrn* is not NULL, it backfills *supported_rtrn* with True if the server supports `DetectableAutorepeat`, and False otherwise. *XkbGetDetectableAutorepeat* returns the current state of `DetectableAutorepeat` for the requesting client: True if `DetectableAutorepeat` is set, and False otherwise.

To set `DetectableAutorepeat`, use *XkbSetDetectableAutorepeat*. This request affects all keyboard activity for the requesting client only; other clients still see the expected non-detectable auto-repeat behavior, unless they have requested otherwise.

```
Bool XkbSetDetectableAutorepeat(display, detectable, supported_rtrn)
    Display * display;          /* connection to X server */
    Bool      detectable;       /* True => set DetectableAutorepeat */
    Bool *     supported_rtrn; /* backfilled True if DetectableAutorepeat supported */
```

XkbSetDetectableAutorepeat sends a request to the server to set `DetectableAutorepeat` on for the current client if *detectable* is True, and off if *detectable* is False; it then waits for a reply. If *supported_rtrn* is not NULL, *XkbSetDetectableAutorepeat* backfills *supported_rtrn* with True if the server supports `DetectableAutorepeat`, and False if it does not. *XkbSetDetectableAutorepeat* returns the current state of `DetectableAutorepeat` for the requesting client: True if `DetectableAutorepeat` is set, and False otherwise.

10.4 Controls for Keyboard Overlays (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 a keyboard that does not actually have one by reusing some portion of the keyboard as an overlay. 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. The two overlays are mutually exclusive; any particular key may be in at most one overlay. `Overlay1` and `Overlay2` are boolean controls. As such, you may enable and disable them using either the `EnabledControls` control or the `AutoReset` control discussed in section 10.1.1.

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 `XkbKB_Overlay1` or `XkbKB_Overlay2` key behaviors, as described in section 16.2 on page 161.

10.5 Controls for Using the Mouse from the Keyboard

Using Xkb, it is possible to configure the keyboard to allow simulation of the X pointer device. This simulation includes both movement of the pointer itself, and press and release events associated with the buttons on the pointer. Two controls affect this behavior: the `MouseKeys` control determines whether or not simulation of the pointer device is active, as well as configuring the default button; the `MouseKeysAccel` control determines the movement characteristics of the pointer when simulated via the keyboard. Both of them are boolean controls; as such, you may enable and disable them using either the `EnabledControls` control or the `AutoReset` control discussed in section 10.1.1. The individual keys which simulate different aspects of the pointer device are determined by the keyboard mapping, discussed in Chapter 16.

10.5.1 The `MouseKeys` Control

The `MouseKeys` control allows a user to 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 `KeyPress` and `KeyRelease` events.

The `MouseKeys` control has a single attribute, `mk_dflt_btn`, which specifies the core button number to be used by mouse keys actions that do not explicitly specify a button. There is no convenience function for getting or setting the attribute; instead call `XkbGetControls` and `XkbSetControls` (see section 10.9 and section 10.10).

Note `MouseKeys` can also be turned on and off by pressing the key combination necessary to produce an `XK_Pointer_EnableKeys` keysym. The defacto default standard for this is `Shift+Alt+NumLock`, but this may vary depending upon the keymap.

10.5.2 The `MouseKeysAccel` Control

When the `MouseKeysAccel` control is enabled, the effect of a key-activated pointer motion action changes as a key is held down. If the control is disabled, pressing a mouse-pointer key yields one mouse event. When `MouseKeysAccel` is enabled, mouse movement is defined by an initial distance specified in the `XkbSA_MovePtr` action and the following fields in the `XkbControlsRec` structure (see section 10.8).

Table 10.2 `MouseKeysAccel` Fields

<i>Field</i>	<i>Function</i>
<code>mk_delay</code>	Time (ms) between the initial key press and the first repeated motion event
<code>mk_interval</code>	Time (ms) between repeated motion events
<code>mk_time_to_max</code>	Number of events (count) before the pointer reaches maximum speed
<code>mk_max_speed</code>	The maximum speed (in pixels per event) the pointer reaches
<code>mk_curve</code>	The ramp used to reach maximum pointer speed

There are no convenience functions to query or change the attributes of the `MouseKeysAccel` control; instead call `XkbGetControls` and `XkbSetControls` (see section 10.9 and section 10.10).

The effects of the attributes of the `MouseKeysAccel` control depend upon whether the `XkbSA_MovePtr` action (see section 16.1) specifies relative or absolute pointer motion.

Absolute Pointer Motion

If an `XkbSA_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. For example, if the `XkbSA_MovePtr` action specifies an absolute position for the X direction, but a relative motion for the Y direction, the pointer accelerates in the Y direction, but stays at the same X position.

Relative Pointer Motion

If the `XkbSA_MovePtr` action specifies relative motion, the initial event always moves the cursor the distance specified in the action. After `mk_delay` milliseconds, a second motion event is generated, and another occurs every `mk_interval` milliseconds until the user releases the key.

Between the time of the second motion event and `mk_time_to_max` intervals, the change in pointer distance per interval increases with each interval. After `mk_time_to_max` intervals have elapsed, the change in pointer distance per interval remains the same, and is calculated by multiplying the original distance specified in the action by `mk_max_speed`.

For example, if the `XkbSA_MovePtr` action specifies a relative motion in the X direction of 5, `mk_delay`=160, `mk_interval`=40, `mk_time_to_max`=30, and `mk_max_speed`=30, the following happens when the user presses the key:

- The pointer immediately moves 5 pixels in the X direction when the key is pressed.
- After 160 milliseconds (`mk_delay`), and every 40 milliseconds thereafter (`mk_interval`), the pointer moves in the X direction.
- The distance in the X direction increases with each interval until 30 intervals (`mk_time_to_max`) have elapsed.
- After 30 intervals, the pointer stops accelerating, and moves 150 pixels (`mk_max_speed * the original distance`) every interval thereafter, until the key is released.

The increase in pointer difference for each interval is a function of `mk_curve`. Events after the first but before maximum acceleration has been achieved are accelerated according to the formula:

$$d(\text{step}) = \text{action_delta} \times \left(\frac{\text{max_accel}}{\text{steps_to_max}^{\text{curveFactor}}} \right) \times \text{step}^{\text{curveFactor}}$$

Where `action_delta` is the relative motion specified by the `XkbSA_MovePtr` action, `mk_max_speed` and `mk_time_to_max` are parameters to the `MouseKeysAccel` control, and the `curveFactor` is computed using the `MouseKeysAccel` `mk_curve` parameter as follows:

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

With the result that a `mk_curve` of 0 causes the distance moved to increase linearly from `action_delta` to (`mk_max_speed * action_delta`). A negative `mk_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 gen-

erated by the action approaches *mk_time_to_max*. The legal values for *mk_curve* are between -1000 and 1000.

A distance vs. time graph of the pointer motion is shown in Figure 10.1.

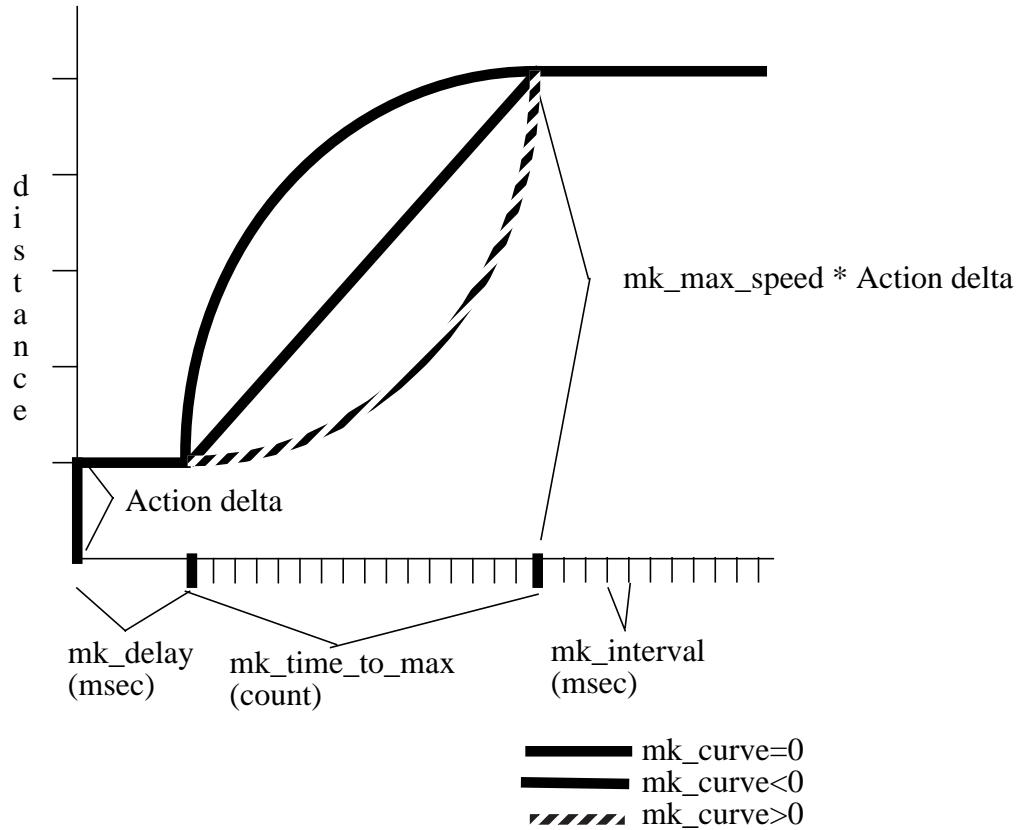


Figure 10.1 MouseKeys Acceleration

10.6 Controls for Better Keyboard Access by Physically-Impaired Persons

The Xkb extension includes several controls specifically aimed at making keyboard use more effective for physically impaired people. All of these controls are boolean controls, and may be individually enabled and disabled, as well as configured to tune their specific behavior. The behavior of these controls is based upon the AccessDOS package¹.

10.6.1 The AccessXKeys Control

Enabling or disabling the keyboard controls through a graphical user interface may be impossible for people who need to use the controls. For example, a user who needs SlowKeys (see section 10.6.6) may not even be able to start the graphical application, let

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.

alone use it, if `SlowKeys` is not enabled. To allow easier access to some of the controls, the `AccessXKeys` control provides a set of special key sequences similar to those available in AccessDOS.

When the `AccessXKeys` control is enabled, the user can turn controls 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 the left or right `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.

When the `AccessXKeys` control is disabled, Xkb does not look for the above special key sequences.

Some of these key sequences optionally generate audible feedback of the change in state, as described in section 10.6.3, or `XkbControlsNotify` events, described in section 10.11.

10.6.2 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 are not accepted until they are held for a certain period of time. To help solve this problem, Xkb provides an `AccessXTIMEOUT` control to automatically change the enabled/disabled state of any boolean controls, and to change the value of the `AccessXKeys` and `AccessXFeedback` control attributes if the keyboard is idle for a specified period of time.

When a timeout as specified by `AccessXTIMEOUT` occurs and a control is consequently modified, Xkb generates an `XkbControlsNotify` event. For more information on `XkbControlsNotify` events, refer to section 10.11 on page 79.

Call `XkbGetAccessXTIMEOUT` to query the current `AccessXTIMEOUT` options for a keyboard device.

```
Bool XkbGetAccessXTIMEOUT(display, device_spec, timeout_rtrn, ctrls_mask_rtrn,
                           ctrls_values_rtrn, options_mask_rtrn, options_values_rtrn)
    Display *           display;          /* connection to X server */
    unsigned int        device_spec;     /* device to query, or XkbUseCoreKbd */
    unsigned short *   timeout_rtrn;    /* delay until AccessXTIMEOUT, seconds */
    unsigned int *     ctrls_mask_rtrn;  /* backfilled with controls to modify */
    unsigned int *     ctrls_values_rtrn; /* backfilled with on/off status for controls */
    unsigned short *   opts_mask_rtrn;   /* backfilled with ax_options to modify */
    unsigned short *   opts_values_rtrn; /* backfilled with values for ax_options */
```

`XkbGetAccessXTIMEOUT` sends a request to the X server to obtain the current values for the `AccessXTIMEOUT` attributes, waits for a reply, and backfills the values into the appropriate arguments. The parameters `opts_mask_rtrn` and `opts_values_rtrn` are backfilled with the options to modify and the values for `ax_options`, which is a field in the `XkbControlsRec` structure (see section 10.8). `XkbGetAccessXTIMEOUT` returns `True` if successful; if a compatible version of the Xkb extension is not available in the server `XkbGetAccessXTIMEOUT` returns `False`.

To configure the AccessXTimeout options for a keyboard device, call *XkbSetAccessXTimeout*.

```
Bool XkbSetAccessXTimeout(display, device_spec, timeout, ctrls_mask, ctrls_values,
                           opts_mask, opts_values)
    Display *           display;      /* connection to X server */
    unsigned int        device_spec;  /* device to configure, or XkbUseCoreKbd */
    unsigned short      timeout;     /* seconds idle until AccessXTimeout occurs */
    unsigned int        ctrls_mask;   /* boolean controls to modify */
    unsigned int        ctrls_values; /* new bits for controls selected by ctrls_mask */
    unsigned short      opts_mask;    /* ax_options to change */
    unsigned short      opts_values;  /* new bits for ax_options selected by opts_mask */
```

timeout specifies the number of seconds the keyboard must be idle before the controls are modified. *ctrls_mask* specifies what controls are to be enabled or disabled, and *ctrls_values* specifies whether those controls are to be enabled or disabled. The bit values correspond to those for enabling and disabling boolean controls (see section 10.1.1). The *opts_mask* field specifies which attributes of the AccessXKeys and AccessXFeedback controls are to be changed, and *opts_values* specifies the new values for those options. The bit values correspond to those for the *ax_options* field of an *XkbDescRec* (see section 10.8).

XkbSetAccessXTimeout sends a request to configure the AccessXTimeout control to the server. It does not wait for a reply, and normally returns True. If a compatible version of the Xkb extension is not available in the server *XkbSetAccessXTimeout* returns False.

10.6.3 The AccessXFeedback Control

Just as some keyboards can produce keyclicks to indicate when a key is pressed or repeating, Xkb can provide feedback for the controls by using special beep codes. Use the AccessXFeedback control to configure the specific types of operations which generate feedback.

There is no convenience function for modifying the AccessXFeedback control, although the feedback as a whole can be enabled or disabled just as other boolean controls are (see section 10.1). Individual beep codes are turned on or off by modifying the following bits in the *ax_options* field of an *XkbControlsRec* structure and using *XkbSetControls* (see section 10.10):

Table 10.3 AccessXFeedback Masks

Action	Beep Code	<i>ax_options</i> bit
LED turned on	High pitched beep	XkbAX_IndicatorFBMask
LED turned off	Low pitched beep	XkbAX_IndicatorFBMask
More than one LED changed state	Two high pitched beeps	XkbAX_IndicatorFBMask
Control turned on	Rising Tone	XkbAX_FeatureFBMask
Control turned off	Falling Tone	XkbAX_FeatureFBMask
More than one control changed state	Two high pitched beeps	XkbAX_FeatureFBMask
SlowKeys and BounceKeys about to be turned on or off	Three high pitched beeps	XkbAX_SlowWarnFBMask
SlowKeys key pressed	Medium pitched beep	XkbAX_SKPressFBMask
SlowKeys key accepted	Medium pitched beep	XkbAX_SKAcceptFBMask
SlowKeys key rejected	Low pitched beep	XkbAX_SKRejectFBMask

Table 10.3 AccessXFeedback Masks

Action	Beep Code	<i>ax_options</i> bit
Accepted SlowKeys key released	Medium pitched beep	XkbAX_SKReleaseFBMask
BounceKeys key rejected	Low pitched beep	XkbAX_BKRejectFBMask
StickyKeys key latched	Low pitched beep followed by high pitched beep	XkbAX_StickyKeysFBMask
StickyKeys key locked	High pitched beep	XkbAX_StickyKeysFBMask
StickyKeys key unlocked	Low pitched beep	XkbAX_StickyKeysFBMask

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. Other implementations can only ring the bell with one fixed pitch. In these cases, use the `XkbAX_DumbBellFBMask` bit of *ax_options* to indicate the bell can only ring with a fixed pitch.

When any of the above feedbacks occur, Xkb may generate a `XkbBellNotify` event (see section 9.4).

10.6.4 AccessXNotify Events

The server can generate `XkbAccessXNotify` events for some of the global keyboard controls. The structure for the `XkbAccessXNotify` event type is as follows:

```
typedef struct {
    int type; /* Xkb extension base event code */
    unsigned long serial; /* X server serial number for event */
    Bool send_event; /* True => synthetically generated */
    Display * display; /* server connection where event generated */
    Time time; /* server time when event generated */
    int xkb_type; /* XkbAccessXNotify */
    int device; /* Xkb device id, will not be XkbUseCoreKbd */
    int detail; /* XkbAXN_* */
    KeyCode keycode; /* key of event */
    int slowKeysDelay; /* current SlowKeys delay */
    int debounceDelay; /* current debounce delay */
} XkbAccessXNotifyEvent;
```

The *detail* field describes what AccessX event just occurred and can be any of the values in Table 10.4.

Table 10.4 AccessXNotify Events

<i>detail</i>	<i>Reason</i>
XkbAXN_SKPress	A key was pressed when SlowKeys was enabled.
XkbAXN_SKAccept	A key was accepted (held longer than the SlowKeys delay).
XkbAXN_SKRelease	An accepted SlowKeys key was released.
XkbAXN_SKReject	A key was rejected (released before the SlowKeys delay expired).
XkbAXN_BKAccept	A key was accepted by BounceKeys.
XkbAXN_BKReject	A key was rejected (pressed before the BounceKeys delay expired).
XkbAXN_AXKWarning	AccessXKeys is about to turn on/off StickyKeys or BounceKeys.

The *keycode* field reports the keycode of the key for which the event occurred. If the action is related to SlowKeys, the *slowKeysDelay* field contains the current SlowKeys acceptance delay. If the action is related to BounceKeys, the *debounceDelay* field contains the current BounceKeys debounce delay.

Selecting for AccessX Events

To receive XkbAccessXNotify events under all possible conditions, call *XkbSelectEvents* (see section 4.3) and pass XkbAccessXNotifyMask in both *bits_to_change* and *values_for_bits*.

To receive XkbStateNotify events only under certain conditions, call *XkbSelectEventDetails* using XkbAccessXNotify as the *event_type* and specifying the desired state changes in *bits_to_change* and *values_for_bits* using mask bits from Table 10.5.

Table 10.5 AccessXNotify Event Details

<i>XkbAccessXNotify Event Details</i>	<i>Value</i>	<i>Circumstances</i>
XkbAXN_SKPressMask	(1<<0)	Slow key press notification wanted
XkbAXN_SKAcceptMask	(1<<1)	Slow key accept notification wanted
XkbAXN_SKRejectMask	(1<<2)	Slow key reject notification wanted
XkbAXN_SKReleaseMask	(1<<3)	Slow key release notification wanted
XkbAXN_BKAcceptMask	(1<<4)	Bounce key accept notification wanted
XkbAXN_BKRejectMask	(1<<5)	Bounce key reject notification wanted
XkbAXN_AXKWarningMask	(1<<6)	AccessX warning notification wanted
XkbAXN_AllEventsMask	(0x7f)	All AccessX features notifications wanted

10.6.5 StickyKeys, RepeatKeys, and MouseKeys Events

The StickyKeys, RepeatKeys, and MouseKeys controls do not generate specific events. Instead, the latching, unlatching, locking or unlocking of modifiers using StickyKeys generates XkbStateNotify events as described in section 5.4. Repeating keys generate normal KeyPress and KeyRelease events, though the auto-repeat can be detected using DetectableAutorepeat (see section 10.3.3). Finally, MouseKeys generates pointer events identical to those of the core pointer device.

10.6.6 The SlowKeys Control

Some users may accidentally bump keys while moving a hand or typing stick toward the key they want. Usually, the keys that are accidentally bumped are just hit 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. Users can then bump any number of keys on their way to the one they want without accidentally getting those characters. Once they have reached the key they want, they can then hold the desired key long enough for the computer to accept it. SlowKeys is a boolean control with one configurable attribute.

When the SlowKeys control is active, the server reports the initial key press, subsequent acceptance or rejection, and release of any key to interested clients by sending an appropriate AccessXNotify event (see section 10.6.4).

To get the SlowKeys acceptance delay for a keyboard device, use *XkbGetSlowKeysDelay*.

```
Bool XkbGetSlowKeysDelay(display, device_spec, delay_rtrn)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int *     delay_rtrn;   /* backfilled with SlowKeys delay, ms */
```

XkbGetSlowKeysDelay requests the attributes of the SlowKeys control from the server, waits for a reply, and backfills *delay_rtrn* with the SlowKeys delay attribute. *XkbGetSlowKeysDelay* returns True if successful; if a compatible version of the Xkb extension is not available in the server, *XkbGetSlowKeysDelay* returns False.

To set the SlowKeys acceptance delay for a keyboard device, call *XkbSetSlowKeysDelay*.

```
Bool XkbSetSlowKeysDelay(display, device_spec, delay)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec; /* device to configure, or XkbUseCoreKbd */
    unsigned int       delay;       /* SlowKeys delay, ms */
```

XkbSetSlowKeysDelay sends a request to configure the SlowKeys control to the server. It does not wait for a reply, and normally returns True. Specifying a value of 0 for the *delay* parameter causes *XkbSetSlowKeys* to generate a BadValue protocol error. If a compatible version of the Xkb extension is not available in the server *XkbSetSlowKeysDelay* returns False.

10.6.7 The BounceKeys Control

Some users may accidentally “bounce” on a key when they release it. They press it once, then accidentally press it again after they release it. The BounceKeys control temporarily disables a key after it has been pressed, effectively “debouncing” the keyboard. The period of time the key is disabled after it is released is known as the *BounceKeys delay*. BounceKeys is a boolean control.

When the BounceKeys control is active, the server reports acceptance or rejection of any key to interested clients by sending an appropriate AccessXNotify event (see section 10.6.4).

Use *XkbGetBounceKeysDelay* to query the current BounceKeys delay for a keyboard device.

```
Bool XkbGetBounceKeysDelay(display, device_spec, delay_rtrn)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int *     delay_rtrn;   /* backfilled with bounce keys delay, ms */
```

XkbGetBounceKeysDelay requests the attributes of the BounceKeys control from the server, waits for a reply, and backfills *delay_rtrn* with the BounceKeys delay attribute. *XkbGetBounceKeysDelay* returns True if successful; if a compatible version of the Xkb extension is not available in the server *XkbGetSlowKeysDelay* returns False.

To set the BounceKeys delay for a keyboard device, use *XkbSetBounceKeysDelay*.

```
Bool XkbSetBounceKeysDelay(display, device_spec, delay)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec;  /* device to configure, or XkbUseCoreKbd */
    unsigned int       delay;       /* bounce keys delay, ms */
```

XkbSetBounceKeysDelay sends a request to configure the BounceKeys control to the server. It does not wait for a reply, and normally returns True. Specifying a value of 0 for the *delay* parameter causes *XkbSetBounceKeysDelay* to generate a BadValue protocol error. If a compatible version of the Xkb extension is not available in the server *XkbSetBounceKeysDelay* returns False.

10.6.8 The StickyKeys Control

Some people find it difficult or even impossible to press two keys at once. For example, a one-fingered typist or someone using a mouth stick cannot press the Shift and 1 keys at the same time. The StickyKeys control solves this problem by changing the behavior of the modifier keys. With StickyKeys, 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, and then press the 1 key.

StickyKeys also allows users to lock modifier keys without requiring special locking keys. When StickyKeys is enabled, a modifier is latched when the user presses it just once. The user can press a modifier twice in a row to lock it, and then unlock it by pressing it one more time.

When a modifier is latched, it becomes unlatched when the user presses a non-modifier key or a pointer button. For instance, to enter the sequence Shift+Control+Z the user could press and release the Shift key to latch it, then press and release the Control key to latch it, and finally press and release the Z key. Because the Control key is a modifier key, pressing it does not unlatch the Shift key. Thus, after the user presses the Control key, both the Shift and Control modifiers are latched. When the user presses the Z key, the effect is as though the user had pressed Shift+Control+Z. In addition, because the Z key is not a modifier key, the Shift and Control modifiers are unlatched.

Locking a modifier key means the modifier affects any key or pointer button the user presses until the user unlocks it or it is unlocked programmatically. For example, to enter the sequence (“XKB”) on a keyboard where ‘(’ is a shifted ‘9’, ‘)’ is a shifted ‘0’, and ‘”’ is a shifted single quote, 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 generates (“XKB”). To unlock the Shift modifier, the user can press and release the Shift key.

StickyKeys is a boolean control with two separate attributes which may be individually configured: one to automatically disable it, and one to control the latching behavior of modifier keys.

StickyKeys Options

The StickyKeys control has two options that can be accessed via the *ax_options* of an *XkbControlsRec* structure (see section 10.8). The first option, TwoKeys, specifies whether StickyKeys should automatically turn off when two keys are pressed at the same time. This feature is useful for shared computers so people who do not want them do

not need to turn StickyKeys off if a previous user left StickyKeys on. The second option, LatchToLock, specifies whether or not StickyKeys locks a modifier when pressed twice in a row.

Call *XkbGetStickyKeysOptions* to query the current StickyKeys attributes for a keyboard device.

```
Bool XkbGetStickyKeysOptions(display, device_spec, options_rtrn)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec;  /* device id, or XkbUseCoreKbd */
    unsigned int *     options_rtrn; /* backfilled with StickyKeys option mask */
```

XkbGetStickyKeysOptions requests the attributes of the StickyKeys control from the server, waits for a reply, and backfills *options_rtrn* with a mask indicating whether the individual StickyKeys options are on or off. Valid bits in *options_rtrn* are:

```
XkbAX_TwoKeysMask
XkbAX_LatchToLockMask
```

XkbGetStickyKeysOptions returns True if successful; if a compatible version of the Xkb extension is not available in the server *XkbGetStickyKeysOptions* returns False.

To set the StickyKeys attributes for a keyboard device, call *XkbSetStickyKeysOptions*.

```
Bool XkbSetStickyKeysOptions(display, device_spec, mask, values)
    Display *          display;      /* connection to X server */
    unsigned int       device_spec;  /* device to configure, or XkbUseCoreKbd */
    unsigned int       mask;        /* selects StickyKeys attributes to modify */
    unsigned int       values;      /* values for selected attributes */
```

XkbSetStickyKeysOptions sends a request to configure the StickyKeys control to the server. It does not wait for a reply, and normally returns True. The valid bits to use for both the *mask* and *values* parameters are:

```
XkbAX_TwoKeysMask
XkbAX_LatchToLockMask
```

If a compatible version of the Xkb extension is not available in the server *XkbSetStickyKeysOptions* returns False.

10.7 Controls for General Keyboard Mapping

There are several controls which apply to the keyboard mapping in general. They control handling of out-of-range group indices, and how modifiers are processed and consumed in the server. These are:

```
GroupsWrap
IgnoreGroupLock
IgnoreLockMods
InternalMods
```

IgnoreGroupLock is a boolean control; the rest are always active.

Without the modifier processing options provided by Xkb, passive grabs set via translations in a client (for example, Alt<KeyPress>space) do not trigger if any modifiers other than those specified by the translation are set. This results in problems in the user

interface when either NumLock or a secondary keyboard group is active. The `IgnoreLockMods` and `IgnoreGroupLock` controls make it possible to avoid this behavior without exhaustively specifying a grab for every possible modifier combination.

10.7.1 The GroupsWrap Control

The GroupsWrap control determines how illegal groups are handled on a global basis. There are a number of valid keyboard sequences which can cause the effective group number to go out of range. When this happens, the group must be normalized back to a valid number. The GroupsWrap control specifies how this is done.

When dealing with group numbers, all computations are done using the group index, which is the group number minus one. There are three different algorithms; the GroupsWrap control specifies which one is used:

- `XkbRedirectInRange`

All invalid group numbers are converted to a valid group number by taking the last four bits of the GroupsWrap control and using them as the group index. If the result is still out of range, Group one is used.

- `XkbClampInRange`

All invalid group numbers are converted to the nearest valid group number. Group numbers larger than the highest supported group number are mapped to the highest supported group; those less than one are mapped to group one.

- `XkbWrapInRange`

All invalid group numbers are converted to a valid group number using integer modulus applied to the group index.

There are no convenience routines for manipulating the GroupsWrap control. Manipulate the GroupsWrap control via the `groups_wrap` field in the `XkbControlsRec` structure, then use `XkbSetControls` and `XkbGetControls` (see section 10.9 and section 10.10) to query and change this control.

Note See also section 15.3.2 on page 134 for a discussion of the related field, `group_info`, which also normalizes a group under certain circumstances.

10.7.2 The IgnoreLockMods Control

The core protocol does not provide a way to exclude specific modifiers from grab calculations, with the result that locking modifiers sometimes have unanticipated side-effects.

The `IgnoreLockMods` control specifies modifiers which should be excluded from grab calculations. These modifiers are also not reported in any core events except `KeyPress` and `KeyRelease` events which do not activate a passive grab and which do not occur while a grab is active.

Manipulate the `IgnoreLockMods` control via the `ignore_lock` field in the `XkbControlsRec` structure, then use `XkbSetControls` and `XkbGetControls` (see section 10.9 and section 10.10) to query and change this control. Alternatively, call `XkbSetIgnoreLockMods`.

To set the modifiers which, if locked, are not to be reported in matching events to passive grabs, call *XkbSetIgnoreLockMods*.

```
Bool XkbSetIgnoreLockMods(display, device_spec, affect_real, real_values, affect_virtual,
                           virtual_values)
    Display * display;      /* connection to the X server */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int affect_real; /* mask of real modifiers affected by this call */
    unsigned int real_values; /* values for affected real modifiers (1=>set, 0=>unset) */
    unsigned int affect_virtual; /* mask of virtual modifiers affected by this call */
    unsigned int virtual_values; /* values for affected virtual modifiers (1=>set, 0=>unset) */
```

XkbSetIgnoreLockMods sends a request to the server to change the server's *IgnoreLockMods* control. *affect_real* and *real_values* are masks of real modifier bits indicating which real modifiers are to be added and removed from the server's *IgnoreLockMods* control. Modifiers selected by both *affect_real* and *real_values* are added to the server's *IgnoreLockMods* control; those selected by *affect_real* but not by *real_values* are removed from the server's *IgnoreLockMods* control. Valid values for *affect_real* and *real_values* consist of any combination of the eight core modifier bits: ShiftMask, LockMask, ControlMask, Mod1Mask - Mod5Mask. *affect_virtual* and *virtual_values* are masks of virtual modifier bits indicating which virtual modifiers are to be added and removed from the server's *IgnoreLockMods* control. Modifiers selected by both *affect_virtual* and *virtual_values* are added to the server's *IgnoreLockMods* control; those selected by *affect_virtual* but not by *virtual_values* are removed from the server's *IgnoreLockMods* control. See section 7.1 on page 31 for a discussion of virtual modifier masks to use in *affect_virtual* and *virtual_values*. *XkbSetIgnoreLockMods* does not wait for a reply from the server. It returns True if the request was sent, and False otherwise.

10.7.3 The IgnoreGroupLock Control

The *IgnoreGroupLock* control is a boolean control with no attributes. If enabled, it specifies that the locked state of the keyboard group should not be considered when activating passive grabs.

Since *IgnoreGroupLock* is a boolean control with no attributes, use the general boolean controls functions (see section 10.1) to change its state.

10.7.4 The InternalMods Control

The core protocol does not provide any means to prevent a modifier from being reported in events sent to clients; Xkb, however makes this possible via the *InternalMods* control. It specifies modifiers which should be consumed by the server and not reported to clients. When a key is pressed and a modifier which has its bit set in the *InternalMods* control is reported to the server, the server uses the modifier when determining the actions to apply for the key. The server then clears the bit, so it is not actually reported to the client. In addition, modifiers specified in the *InternalMods* control are not used to determine grabs and are not used to calculate core protocol compatibility state.

Manipulate the *InternalMods* control via the *internal* field in the *XkbControlsRec* structure, using *XkbSetControls* and *XkbGetControls* (see section 10.9 and section 10.10). Alternatively, call *XkbSetServerInternalMods*.

To set the modifiers which are consumed by the server before events are delivered to the client, call *XkbSetServerInternalMods*.

```
Bool XkbSetServerInternalMods(display, device_spec, affect_real, real_values, affect_virtual,
                               virtual_values)
    Display * display;      /* connection to the X server */
    unsigned int device_spec; /* device id, or XkbUseCoreKbd */
    unsigned int affect_real; /* mask of real modifiers affected by this call */
    unsigned int real_values; /* values for affected real modifiers (1=>set, 0=>unset) */
    unsigned int affect_virtual; /* mask of virtual modifiers affected by this call */
    unsigned int virtual_values; /* values for affected virtual modifiers (1=>set, 0=>unset) */
```

XkbSetServerInternalMods sends a request to the server to change the internal modifiers consumed by the server. *affect_real* and *real_values* are masks of real modifier bits indicating which real modifiers are to be added and removed from the server's internal modifiers control. Modifiers selected by both *affect_real* and *real_values* are added to the server's internal modifiers control; those selected by *affect_real* but not by *real_values* are removed from the server's internal modifiers mask. Valid values for *affect_real* and *real_values* consist of any combination of the eight core modifier bits: ShiftMask, LockMask, ControlMask, Mod1Mask - Mod5Mask. *affect_virtual* and *virtual_values* are masks of virtual modifier bits indicating which virtual modifiers are to be added and removed from the server's internal modifiers control. Modifiers selected by both *affect_virtual* and *virtual_values* are added to the server's internal modifiers control; those selected by *affect_virtual* but not by *virtual_values* are removed from the server's internal modifiers control. See section 7.1 on page 31 for a discussion of virtual modifier masks to use in *affect_virtual* and *virtual_values*. *XkbSetServerInternalMods* does not wait for a reply from the server. It returns True if the request was sent, and False otherwise.

10.8 The XkbControlsRec Structure

Many of the individual controls described in sections 10.1 through 10.7 may be manipulated via convenience routines discussed in those sections. Some of them, however, have no convenience routines. The *XkbControlsRec* structure allows the manipulation of one or more of the controls in a single operation, and to track changes to any of them, in conjunction with the *XkbGetControls* and *XkbSetControls* functions. This is the only way to manipulate those controls which have no convenience functions.

The *XkbControlsRec* structure is defined as follows:

```
#define XkbMaxLegalKeyCode      255
#define XkbPerKeyBitArraySize   ((XkbMaxLegalKeyCode+1)/8)

typedef struct {
    unsigned char    mk_dflt_btn;      /* default button for keyboard driven mouse */
    unsigned char    num_groups;      /* number of keyboard groups */
    unsigned char    groups_wrap;     /* how to wrap out-of-bounds groups */
    XkbModsRec      internal;        /* defines server internal modifiers */
    XkbModsRec      ignore_lock;     /* modifiers to ignore when checking for grab */
    unsigned int     enabled_ctrls;   /* 1 bit => corresponding boolean control enabled */
    unsigned short   repeat_delay;    /* ms delay until first repeat */
    unsigned short   repeat_interval; /* ms delay between repeats */
    unsigned short   slow_keys_delay; /* ms minimum time key must be down to be ok */
    unsigned short   debounce_delay;  /* ms delay before key reactivated */
```

```

        unsigned short    mk_delay;      /* ms delay to second mouse motion event */
        unsigned short    mk_interval;   /* ms delay between repeat mouse events */
        unsigned short    mk_time_to_max; /* # intervals until constant mouse move */
        unsigned short    mk_max_speed;  /* multiplier for maximum mouse speed */
        short             mk_curve;     /* determines mouse move curve type */
        unsigned short    ax_options;    /* 1 bit => Access X option enabled */
        unsigned short    ax_timeout;   /* seconds until Access X disabled */
        unsigned short    axt_opts_mask; /* 1 bit => options to reset on Access X timeout */
        unsigned short    axt_opts_values; /* 1 bit => turn option on, 0=> off */
        unsigned int      axt_ctrls_mask; /* which bits in enabled_ctrls to modify */
        unsigned int      axt_ctrls_values; /* values for new bits in enabled_ctrls */
        unsigned char     per_key_repeat[XkbPerKeyBitArraySize]; /* per key auto repeat */

} XkbControlsRec, *XkbControlsPtr;

```

The general purpose routines which work with the `XkbControlsRec` structure use a mask to specify which controls are to be manipulated. Table 10.6 lists these controls, the masks used to select them in the general function calls (*which* parameter), and the data fields in the `XkbControlsRec` structure which comprise each of the individual controls. Also listed is the bit used to turn boolean controls on and off, and the section where each control is described in more detail.

Table 10.6 Xkb Controls]

<i>Control</i>	<i>Control Selection Mask</i> (<i>which</i> parameter)	<i>Relevant XkbControlsRec</i> <i>Data Fields</i>	<i>Boolean Control</i> <i>enabled_ctrls</i> bit	<i>Section</i>
AccessXFeedback	XkbAccessXFeedbackMask	ax_options: XkbAX_*FBMask		XkbAccessXFeedbackMask 10.6.3
AccessXKeys			XkbAccessXKeysMask	10.6.1
AccessXTIMEOUT	XkbAccessXTIMEOUTMask	ax_timeout axt_opts_mask axt_opts_values axt_ctrls_mask axt_ctrls_values	XkbAccessXTIMEOUTMask	10.6.2
AudibleBell			XkbAudibleBellMask	9.2
AutoReset				10.1.2
BounceKeys	XkbBounceKeysMask	debounce_delay	XkbBounceKeysMask	10.6.7
Detectable- Autorepeat				10.3.3
EnabledControls	XkbControlsEnabledMask	enabled_ctrls		Non-Boolean Control 10.1.1
GroupsWrap	XkbGroupsWrapMask	groups_wrap		Non-Boolean Control 10.7.1
IgnoreGroupLock			XkbIgnoreGroupLockMask	10.7.3
IgnoreLockMods	XkbIgnoreLockModsMask	ignore_lock		Non-Boolean Control 5.1
InternalMods	XkbInternalModsMask	internal		Non-Boolean Control 5.1
MouseKeys	XkbMouseKeysMask	mk_dflt_btn	XkbMouseKeysMask	10.5.1
MouseKeysAccel	XkbMouseKeysAccelMask	mk_delay mk_interval mk_time_to_max mk_max_speed mk_curve	XkbMouseKeysAccelMask	10.5.2
Overlay1			XkbOverlay1Mask	10.4

Table 10.6 Xkb Controls]

<i>Control</i>	<i>Control Selection Mask (which parameter)</i>	<i>Relevant XkbControlsRec Data Fields</i>	<i>Boolean Control enabled_ctrls bit</i>	<i>Section</i>
Overlay2			XkbOverlay2Mask	10.4
PerKeyRepeat	XkbPerKeyRepeatMask	per_key_repeat	<i>Non-Boolean Control</i>	10.3.1
RepeatKeys	XkbRepeatKeysMask	repeat_delay repeat_interval	XkbRepeatKeysMask	10.3
SlowKeys	XkbSlowKeysMask	slow_keys_delay	XkbSlowKeysMask	10.6.6
StickyKeys	XkbStickyKeysMask	ax_options: XkbAX_TwoKeysMask XkbAX_LatchToLockMask	XkbStickyKeysMask	10.6.8

Table 10.7 shows the actual values for the individual mask bits used to select controls for modification, and to enable and disable the control. Note that the same mask bit is used to specify general modifications to the parameters used to configure the control (*which*), and to enable and disable the control (*enabled_ctrls*). The anomalies in the table (no “ok” in column) are for controls which have no configurable attributes; and for controls which are not boolean controls, and therefore cannot be enabled or disabled.

Table 10.7 Controls Mask Bits

<i>Mask Bit</i>	<i>which or changed_ctrls</i>	<i>enabled_ctrls</i>	<i>Value</i>
XkbRepeatKeysMask	ok	ok	(1L<<0)
XkbSlowKeysMask	ok	ok	(1L<<1)
XkbBounceKeysMask	ok	ok	(1L<<2)
XkbStickyKeysMask	ok	ok	(1L<<3)
XkbMouseKeysMask	ok	ok	(1L<<4)
XkbMouseKeysAccelMask	ok	ok	(1L<<5)
XkbAccessXKeysMask	ok	ok	(1L<<6)
XkbAccessXTimeoutMask	ok	ok	(1L<<7)
XkbAccessXFeedbackMask	ok	ok	(1L<<8)
XkbAudibleBellMask		ok	(1L<<9)
XkbOverlay1Mask		ok	(1L<<10)
XkbOverlay2Mask		ok	(1L<<11)
XkbIgnoreGroupLockMask		ok	(1L<<12)
XkbGroupsWrapMask	ok		(1L<<27)
XkbInternalModsMask	ok		(1L<<28)
XkbIgnoreLockModsMask	ok		(1L<<29)
XkbPerKeyRepeatMask	ok		(1L<<30)
XkbControlsEnabledMask	ok		(1L<<31)
XkbAccessXOptionsMask	ok	ok	(XkbStickyKeysMask XkbAccessXFeedbackMask)
XkbAllBooleanCtrlsMask		ok	(0x00001FFF)
XkbAllControlsMask	ok		(0xF8001FFF)

The individual fields of the XkbControlsRec structure are described below.

mk_dflt_btn

mk_dflt_btn is an attribute of the MouseKeys control (see section 10.5). It specifies the mouse button number to use for keyboard simulated mouse button operations. Its value should be one of the core symbols Button1 - Button5.

num_groups

num_groups is not a part of any control, but is reported in the XkbControlsRec structure whenever any of its components are fetched from the server. It reports the number of groups the particular keyboard configuration uses, and is computed automatically by the server whenever the keyboard mapping changes.

groups_wrap

groups_wrap is an attribute of the GroupsWrap control (see section 10.7.1). It specifies the handling of illegal groups on a global basis. Valid values for *groups_wrap* are shown in Table 10.8.

Table 10.8 GroupsWrap options (*groups_wrap* field)

<i>groups_wrap symbolic name</i>	<i>value</i>
XkbWrapInRange	(0x00)
XkbClampInRange	(0x40)
XkbRedirectInRange	(0x80)

When *groups_wrap* is set to XkbRedirectInRange, its four low order bits specify the index of the group to use.

internal

internal is an attribute of the InternalMods control (see section 10.7.4). It specifies modifiers to be consumed in the server and not passed on to clients when events are reported. Valid values consist of any combination of the eight core modifier bits: ShiftMask, LockMask, ControlMask, Mod1Mask - Mod5Mask.

ignore_lock

ignore_lock is an attribute of the IgnoreLockMods control (see section 10.7.2). It specifies modifiers to be ignored in grab calculations. Valid values consist of any combination of the eight core modifier bits: ShiftMask, LockMask, ControlMask, Mod1Mask - Mod5Mask.

enabled_ctrls

enabled_ctrls is an attribute of the EnabledControls control (see section 10.1.1). It contains one bit per boolean control. Each bit determines whether the corresponding control is enabled or disabled; a one bit means the control is enabled. The mask bits used to enable these controls are listed in Table 10.7, using only those masks with “ok” in the *enabled_ctrls* column.

repeat_delay and repeat_interval

repeat_delay and *repeat_interval* are attributes of the RepeatKeys control (see section 10.3.2). *repeat_delay* is the initial delay before a key begins repeating, in milliseconds; *repeat_interval* is the delay between subsequent key events, in milliseconds.

slow_keys_delay

slow_keys_delay is an attribute of the SlowKeys control (see section 10.6.6). Its value specifies the SlowKeys acceptance delay period in milliseconds before a key press is accepted by the server.

debounce_delay

debounce_delay is an attribute of the BounceKeys control (see section 10.6.7). Its value specifies the BounceKeys delay period in milliseconds for which the key is disabled after having been pressed before another press of the same key is accepted by the server.

mk_delay, mk_interval, mk_time_to_max, mk_max_speed, and mk_curve

mk_delay, *mk_interval*, *mk_time_to_max*, *mk_max_speed*, and *mk_curve* are attributes of the MouseKeysAccel control. Refer to section 10.5.2 for a description of these fields and the units involved.

ax_options

The *ax_options* field contains attributes used to configure two different controls, the StickyKeys control (see section 10.6.8) and the AccessXFeedback control (see section 10.6.3). The *ax_options* field is a bitmask, and may include any combination of the bits defined in Table 10.9.

Table 10.9 Access X Enable/Disable Bits (*ax_options* field)

<i>Access X Control</i>	<i>ax_options bit</i>	<i>value</i>
AccessXFeedback	XkbAX_SKPressFBMask	(1L<<0)
	XkbAX_SKAcceptFBMask	(1L << 1)
	XkbAX_FeatureFBMask	(1L << 2)
	XkbAX_SlowWarnFBMask	(1L << 3)
	XkbAX_IndicatorFBMask	(1L << 4)
	XkbAX_StickyKeysFBMask	(1L << 5)
	XkbAX_SKReleaseFBMask	(1L << 8)
	XkbAX_SKRejectFBMask	(1L << 9)
	XkbAX_BKRejectFBMask	(1L << 10)
	XkbAX_DumbBellFBMask	(1L << 11)
	XkbAX_TwoKeysMask	(1L << 6)
	XkbAX_LatchToLockMask	(1L << 7)
StickyKeys	XkbAX_AllOptionsMask	(0xFFFF)

The fields pertaining to each control are relevant only when the control is enabled (XkbAccessXFeedbackMask or XkbStickyKeysMask bit is turned on in the *enabled_cntrls* field).

Xkb provides a set of convenience macros for working with the *ax_options* field of an XkbControlsRec structure:

```
#define XkbAX_NeedOption(c,w) ((c)->ax_options&(w))
```

The *XkbAX_NeedOption* macro is useful for determining if a particular AccessX option is enabled or not. It accepts a pointer to an XkbControlsRec structure and a valid mask bit

from Table 10.9. If the specified mask bit in the *ax_options* field of the controls structure is set, the macro returns the mask bit. Otherwise, it returns zero. Thus,

`XkbAX_NeedOption(ctlrec, XkbAX_LatchToLockMask)`

is non-zero if the latch to lock transition for latching keys is enabled, and zero if it is disabled. Note that *XkbAX_NeedOption* only determines whether or not the particular capability is configured to operate; the `XkbAccessXFeedbackMask` bit must also be turned on in *enabled_ctrls* for the capability to actually be functioning.

```
#define XkbAX_AnyFeedback(c) ((c)->enabled_ctrls&XkbAccessXFeedbackMask)
```

The *XkbAX_AnyFeedback* macro accepts a pointer to an `XkbControlsRec` structure and tells whether the `AccessXFeedback` control is enabled or not. If the `AccessXFeedback` control is enabled, the macro returns `XkbAccessXFeedbackMask`. Otherwise, it returns zero.

```
#define XkbAX_NeedFeedback(c,w)  
      (XkbAX_AnyFeedback(c)&&XkbAX_NeedOption(c,w))
```

The *XkbAX_NeedFeedback* macro is useful for determining if both the `AccessXFeedback` control and a particular AccessX feedback option are enabled. The macro accepts a pointer to an `XkbControlsRec` structure and a feedback option from the table above. If both the `AccessXFeedback` control and the specified feedback option are enabled, the macro returns True. Otherwise it returns False.

ax_timeout, axt_opts_mask, axt_opts_values, axt_ctrls_mask, and axt_ctrls_values

ax_timeout, *act_opts_mask*, *axt_opts_values*, *axt_ctrls_mask*, and *axt_ctrls_values* are attributes of the `AccessXTimeout` control. Refer to section 10.6.2 for a description of these fields and the units involved.

per_key_repeat

The *per_key_repeat* field mirrors the *auto_repeats* field of the core protocol `XKeyboardState` structure: changing the *auto_repeats* field automatically changes *per_key_repeat* and vice-versa. It is provided for convenience and to reduce protocol traffic. For example, to obtain the individual repeat key behavior as well as the repeat delay and rate, call *XkbGetControls*. If the *per_key_repeat* were not in this structure, you would have to call both *XGetKeyboardControl* and *XkbGetControls* to get this information. The bits correspond to keycodes. The first seven keys (keycodes 1-7) are indicated in *per_key_repeat[0]*, with bit position 0 (low order) corresponding to the fictitious keycode 0. Following array elements correspond to 8 keycodes per element. A 1 bit indicates the key is a repeating key.

10.9 Querying Controls

Call *XkbGetControls* to find the current state of Xkb server controls.

```
Status XkbGetControls(display, which, xkb)
    Display *          display;      /* connection to X server */
    unsigned long      which;       /* mask of controls requested */
    XkbDescPtr         xkb;        /* keyboard description for controls information*/
```

XkbGetControls queries the server for the requested control information, waits for a reply, and then copies the server's values for the requested information into the *ctrls* structure of the *xkb* argument. Only those components specified by the *which* parameter are copied. Valid values for *which* are any combination of the masks listed in Table 10.7 on page 74 that have "ok" in the *which* column.

If *xkb->ctrls* is NULL, *XkbGetControls* allocates and initializes it before obtaining the values specified by *which*. If *xkb->ctrls* is not NULL, *XkbGetControls* modifies only those portions of *xkb->ctrls* corresponding to the values specified by *which*.

XkbGetControls returns Success if successful; otherwise it returns BadAlloc if it cannot obtain sufficient storage, BadMatch if *xkb* is NULL or *which* is empty, or BadImplementation.

To free the *ctrls* member of a keyboard description, use *XkbFreeControls* (see section 10.12)

The *num_groups* field in the *ctrls* structure is always filled in by *XkbGetControls*, regardless of which bits are selected by *which*.

10.10 Changing Controls

There are two ways to make changes to controls: either change a local copy keyboard description and call *XkbSetControls*, or, to reduce network traffic, use an *XkbControlsChangesRec* structure and call *XkbChangeControls*.

To change the state of one or more controls, first modify the *ctrls* structure in a local copy of the keyboard description and then use *XkbSetControls* to copy those changes to the X server.

```
Bool XkbSetControls(display, which, xkb)
    Display *          display;      /* connection to X server */
    unsigned long      which;       /* mask of controls to change */
    XkbDescPtr         xkb;        /* ctrls field contains new values to be set */
```

For each bit that is set in the *which* parameter, *XkbSetControls* sends the corresponding values from the *xkb->ctrls* field to the server. Valid values for *which* are any combination of the masks listed in Table 10.7 on page 74 that have "ok" in the *which* column.

If *xkb->ctrls* is NULL, the server does not support a compatible version of Xkb, or the Xkb extension has not been properly initialized, *XkbSetControls* returns False. Otherwise, it sends the request to the X server and returns True.

Note that changes to attributes of controls in the *XkbControlsRec* structure are only apparent when the associated control is enabled, although the corresponding values are still updated in the X server. For example, the *repeat_delay* and *repeat_interval* fields are ignored unless the RepeatKeys control is enabled (that is, the X server's equivalent of *xkb->ctrls* has *XkbRepeatKeyMask* set in *enabled_ctrls*). It is permissible to modify the attributes of a control in one call to *XkbSetControls*, and enable the control in a subsequent call. See section 10.1.1 for more information on enabling and disabling controls.

Note that the *enabled_ctrls* field is itself a control — the *EnabledControls* control. As such, to set a specific configuration of enabled and disabled boolean controls, you must set *enabled_ctrls* to the appropriate bits to enable only the controls you want and disable all others, then specify the *XkbControlsEnabledMask* in a call to *XkbSetControls*. Since

this is somewhat awkward if all you want to do is enable and disable controls, and not modify any of their attributes, a convenience function is also provided for this purpose (*XkbChangeEnabledControls*, section 10.1.1).

10.10.1 The **XkbControlsChangesRec** Structure

The **XkbControlsChangesRec** structure allows applications to track modifications to an **XkbControlsRec** structure and thereby reduce the amount of traffic sent to the server. The same **XkbControlsChangesRec** structure may be used in several successive modifications to the same **XkbControlsRec** structure, then subsequently used to cause all of the changes, and only the changes, to be propagated to the server. The **XkbControlsChangesRec** structure is defined below:

```
typedef struct _XkbControlsChanges {
    unsigned int changed_ctrls;           /* bits indicating changed control data */
    unsigned int enabled_ctrls_changes;   /* bits indicating enabled/disabled controls */
    Bool      num_groups_changed;        /* True if number of keyboard groups changed */
} XkbControlsChangesRec,*XkbControlsChangesPtr;
```

The *changed_ctrls* field is a mask specifying which logical sets of data in the controls structure have been modified. In this context, modified means *set*, that is, if a value is set to the same value it previously contained, it has still been modified, and is noted as changed. Valid values for *changed_ctrls* are any combination of the masks listed in Table 10.7 on page 74 that have “ok” in the *changed_ctrls* column. Setting a bit implies the corresponding data fields from the “Relevant XkbControlsRec Data Fields” column in Table 10.6 have been modified. The *enabled_ctrls_changes* field specifies which bits in the *enabled_ctrls* field have changed. If the number of keyboard groups has changed, the *num_groups_changed* field is set to True.

If you have an Xkb description with controls which have been modified, and an **XkbControlsChangesRec** which describes the changes which have been made, the *XkbChangeControls* function provides a flexible method for updating the controls in a server to match those in the changed keyboard description.

```
Bool XkbChangeControls(dpy, xkb, changes)
    Display *          dpy;      /* connection to X server */
    XkbDescPtr         xkb;      /* keyboard description with changed xkb->ctrls */
    XkbControlsChangesPtr changes; /* which parts of xkb->ctrls have changed */
```

XkbChangeControls copies any controls fields specified by *changes* from the keyboard description controls structure, *xkb->ctrls*, to the server specified by *dpy*.

10.11 Tracking Changes to Keyboard Controls

Whenever a field in the controls structure changes in the server’s keyboard description, the server sends an **XkbControlsNotify** event to all interested clients. To receive **XkbControlsNotify** events under all possible conditions, call *XkbSelectEvents* (see section 4.3) and pass **XkbControlsNotifyMask** in both *bits_to_change* and *values_for_bits*.

To receive **XkbControlsNotify** events only under certain conditions, call *XkbSelectEventDetails* using **XkbControlsNotify** as the *event_type* and specifying the desired state changes in *bits_to_change* and *values_for_bits* using mask bits from Table 10.7 on page 74.

The structure for the `XkbControlsNotify` event is defined as follows:

```
typedef struct {
    int             type;          /* Xkb extension base event code */
    unsigned long   serial;        /* X server serial number for event */
    Bool            send_event;    /* True => synthetically generated */
    Display *       display;       /* server connection where event generated */
    Time            time;          /* server time when event generated */
    int             xkb_type;      /* XkbCompatMapNotify */
    int             device;        /* Xkb device id, will not be XkbUseCoreKbd */
    unsigned int    changed_ctrls; /* bits indicating which controls data have changed*/
    unsigned int    enabled_ctrls; /* controls currently enabled in server */
    unsigned int    enabled_ctrl_changes; /* bits indicating enabled/disabled controls */
    int             num_groups;   /* current number of keyboard groups */
    KeyCode         keycode;       /* != 0 => keycode of key causing change */
    char            event_type;    /* Type of event causing change */
    char            req_major;     /* major event code of event causing change */
    char            req_minor;     /* minor event code of event causing change */
} XkbControlsNotifyEvent;
```

The `changed_ctrls` field specifies the controls components that have changed and consists of bits taken from the masks defined in Table 10.7 on page 74, with “ok” in the `changed_ctrls` column.

The controls currently enabled in the server are reported in the `enabled_ctrls` field. If any controls were just enabled or disabled (that is, the contents of the `enabled_ctrls` field changed), they are flagged in the `enabled_ctrl_changes` field. The valid bits for these fields are the masks listed in Table 10.7, with “ok” in the `enabled_ctrls` column. The `num_groups` field reports the number of groups bound to the key belonging to the most number of groups, and is automatically updated when the keyboard mapping changes.

If the change was caused by a request from a client, the `keycode` and `event_type` fields are set to 0 and the `req_major` and `req_minor` fields identify the request. The `req_major` value is the same as the major extension opcode. Otherwise, `event_type` is set to the type of event that caused the change (one of KeyPress, KeyRelease, DeviceKeyPress, DeviceKeyRelease, ButtonPress or ButtonRelease), and `req_major` and `req_minor` are undefined. If `event_type` is KeyPress, KeyRelease, DeviceKeyPress, or DeviceKeyRelease, the `keycode` field is set to the key that caused the change. If `event_type` is ButtonPress or ButtonRelease, `keycode` contains the button number.

When a client receives an `XkbControlsNotify` event, it can note the changes in a changes structure using `XkbNoteControlsChanges`.

```
void XkbNoteControlsChanges(changes, new, wanted)
    XkbControlsChangesPtr   changes; /* records changes indicated by new */
    XkbControlsNotifyEvent * new;    /* tells which things have changed */
    unsigned int             wanted; /* tells which parts of new to record in changes */
```

The `wanted` parameter is a bitwise inclusive OR of bits taken from the set of masks specified in Table 10.7, with “ok” in the `changed_ctrls` column. `XkbNoteControlsChanges` copies any changes reported in `new` and specified in `wanted` into the changes record specified by `old`.

Use *XkbGetControlsChanges* to update a local copy of a keyboard description with the changes previously noted by one or more calls to *XkbNoteControlsChanges*.

Status **XkbGetControlsChanges**(*dpy*, *xkb*, *changes*)

Display *	<i>dpy</i> ;	/* connection to X server */
XkbDescPtr	<i>xkb</i> ;	/* <i>xkb->ctrls</i> will be updated */
XkbNameChangesPtr	<i>changes</i> ;	/* indicates which parts of <i>xkb->ctrls</i> to update */

XkbGetControlsChanges examines the *changes* parameter, queries the server for the necessary information, and copies the results into the *xkb->ctrls* keyboard description. If the *ctrls* field of *xkb* is NULL, *XkbGetControlsChanges* allocates and initializes it. To free the *ctrls* field, use *XkbFreeControls* (see section 10.12).

XkbGetControlsChanges returns Success if successful, and can generate BadAlloc, BadImplementation and BadMatch errors.

10.12 Allocating and Freeing an XkbControlsRec

The need to allocate an XkbControlsRec structure seldom arises; Xkb creates one when an application calls *XkbGetControls* or a related function. For those situations where there is not an XkbControlsRec structure allocated in the XkbDescRec, allocate one by calling *XkbAllocControls*.

Status **XkbAllocControls**(*xkb*, *which*)

XkbDescPtr	<i>xkb</i> ;	/* Xkb description in which to allocate ctrls rec */
unsigned int	<i>which</i> ;	/* mask of components of <i>ctrls</i> to allocate */

XkbAllocControls allocates the *ctrls* field of the *xkb* parameter, initializes all fields to 0, and returns Success. If the *ctrls* field is not NULL, *XkbAllocControls* simply returns Success. If *xkb* is NULL, *XkbAllocControls* reports a BadMatch error. If the *ctrls* field could not be allocated, it reports a BadAlloc error.

The *which* mask specifies the individual fields of the *ctrls* structure to be allocated and can contain any of the valid masks defined in Table 10.7. Since none of the currently existing controls have any structures associated with them, *which* is currently of little practical value in this call.

To free memory used by the *ctrls* member of an XkbDescRec structure, use *XkbFreeControls*:

void **XkbFreeControls**(*xkb*, *which*, *free_all*)

XkbDescPtr	<i>xkb</i> ;	/* Xkb description in which to free controls components */
unsigned int	<i>which</i> ;	/* mask of components of <i>ctrls</i> to free */
Bool	<i>free_all</i> ;	/* True => free everything + ctrls itself */

XkbFreeControls frees the specified components of the *ctrls* field in the *xkb* keyboard description, and sets the corresponding structure component values to NULL or 0. The *which* mask specifies the fields of *ctrls* to be freed and can contain any of the controls components specified in Table 10.7.

If *free_all* is True, *XkbFreeControls* frees every non-NULL structure component in the controls, frees the XkbControlsRec structure referenced by the *ctrls* member of *xkb*, and sets *ctrls* to NULL.

11 X Library Controls

The Xkb extension is composed of two parts: a server extension, and a client-side X library extension. Chapter 10 discusses functions used to modify controls effecting the behavior of the server portion of the Xkb extension. This chapter discusses functions used to modify controls which effect only the behavior of the client portion of the extension; these controls are known as Library Controls.

All of the Library Controls are boolean flags which may be enabled and disabled. The controls can be divided into several categories:

- controls affecting general string lookups
- controls affecting compose processing
- controls affecting event delivery

There are two types of string lookups performed by *XLookupString*. The first type involves translating a single keycode into a string; the controls in the first category affect this type of lookup. The second type involves translating a series of keysyms into a string; the controls in the second category affect this type of lookup.

An Xkb implementation is required to support the programming interface for all of the controls. However, an implementation may choose not to support the semantics associated with the controls which deal with compose processing. In this case, a program which accesses these controls should still function normally; however, the feedback which would normally occur with the controls enabled may be missing.

11.1 Controls Affecting Keycode-to-String Translation

The first type of string lookups, which are here called *simple string lookups*, involves translating a single keycode into a string. Because these simple lookups involve only a single keycode, all of the information needed to do the translation is contained in the keyboard state in a single event. The controls affecting simple string lookups are:

```
ForceLatin1Lookup  
ConsumeLookupMods  
LevelOneUsesShiftAndLock
```

11.1.1 ForceLatin1Lookup

If the ForceLatin1Lookup control is enabled, *XLookupString* only returns strings using the Latin1 character set. If ForceLatin1Lookup is not enabled, *XLookupString* can return characters which are not in the Latin1 set. By default, this control is disabled, allowing characters outside of the Latin1 set to be returned.

11.1.2 ConsumeLookupMods

Simple string lookups in *XLookupString* involve two different translation phases. The first phase translates raw device keycodes to individual keysyms. The second phase attempts to map the resulting keysym into a string of one or more characters. In the first phase, some of the modifiers are normally used to determine the appropriate shift level for a key.

The ConsumeLookupMods control determines whether or not *XLookupString* consumes the modifiers it uses during the first phase of processing (mapping a keycode to a keysym). When a modifier is consumed, it is effectively removed from the working copy of

the keyboard state information *XLookupString* is using, and appears to be unset for the remainder of the processing.

If the `ConsumeLookupMods` control is enabled, *XLookupString* does not use the modifiers used to translate the keycode of the event to a keysym when it is determining the string associated with a keysym. For example, assume the keymap for the ‘A’ key only contains the shift modifier and the `ConsumeLookupMods` control is enabled. If a user presses the Shift key and the A key while the Num_Lock key is locked, *XLookupString* uses the Shift modifier when mapping the keycode for the ‘a’ key to the keysym for ‘A’; subsequently, it only uses the NumLock modifier when determining the string associated with the keysym ‘A’.

If the `ConsumeLookupMods` control is not enabled, *XLookupString* uses all of the event modifiers to determine the string associated with a keysym. This behavior mirrors the behavior of *XLookupString* in the core implementation.

The `ConsumeLookupMods` control is unset by default. For more information on modifier consumption, refer to Chapter 12.

11.1.3 AlwaysConsumeShiftAndLock

The `AlwaysConsumeShiftAndLock` control, if enabled, forces *XLookupString* to consume the Shift and Lock modifiers when processing all keys, even if the definition for the key type does not specify these modifiers. The `AlwaysConsumeShiftAndLock` control is unset by default. See section 15.2 on page 127 for a discussion of key types.

11.2 Controls Affecting Compose Processing

The second type of string lookup performed by *XLookupString* involves translating a series of keysyms into a string. Because these lookups can involve more than one key event, they require *XLookupString* to retain some state information between successive calls. The process of mapping a series of keysyms to a string is known as *compose processing*. The controls affecting compose processing are:

```
ConsumeKeysOnComposeFail  
ComposeLED  
BeepOnComposeFail
```

Because different vendors have historically used different algorithms to implement compose processing, and these algorithms may be incompatible with the semantics required by the Xkb compose processing controls, implementation of the compose processing controls is optional in an Xkb implementation.

11.2.1 ConsumeKeysOnComposeFail

Some compose processing algorithms signal the start of a compose sequence by a key event meaning “start compose”.¹ The subsequent key events should normally result in a valid composition yielding a valid translation to a string. If the subsequent key events do not have a valid translation, some decision must be made about what to do with the key events which were processed while attempting the compose. The `ConsumeKeysOnCom-`

1. Another possibility is to have the compose processing simply be the result of a finite state acceptor; a compose sequence would never fail for a properly written finite state acceptor.

`poseFail` control allows a client to specify what happens with the key events *XLookupString* has been considering when it reaches a dead end in a compose sequence.

If the `ConsumeKeysOnComposeFail` control is set, all keys associated with a failed compose sequence should be consumed (discarded). If the `ConsumeKeysOnComposeFail` control is not set, the key events associated with a failed compose sequence should be processed as a normal sequence of key events.

The `ConsumeKeysOnComposeFail` control is disabled by default.

11.2.2 ComposeLED

The `ComposeLED` control allows a client to specify whether or not an indicator should be set and cleared to provide feedback when compose processing is in progress. The control does not specify which indicator should be used; the mapping for this is up to the individual implementation. If the `ComposeLED` control is enabled, it specifies that an indicator should be set when a compose sequence is in progress, and cleared when one is not in progress. The `ComposeLED` control is disabled by default.

While the Xkb extension does not specify the type of indicator to be used when the `ComposeLED` control is implemented, a consistent convention between implementations is to everyone's benefit. If a named indicator is used for this purpose, the recommended name is "Compose". Note that some implementations may use an unnamed, custom hardware LED for this purpose.

11.2.3 BeepOnComposeFail

The `BeepOnComposeFail` control allows a client to specify whether or not a bell should be activated to provide feedback when a compose sequence fails. The control does not specify the type of bell which should be used; the mapping for this is up to the individual implementation. If the `BeepOnComposeFail` control is enabled, it specifies that a bell should be activated when a compose sequence fails. The `BeepOnComposeFail` control is disabled by default. If implemented, the bell should be activated using *XkbBell* or *XkbDeviceBell*.

While the Xkb extension does not specify the type of bell to be used when the `BeepOnComposeFail` control is implemented, a consistent convention between implementations is to everyone's benefit. If a named bell is used for this purpose, the recommended name is "ComposeFail".

11.3 Controls Effecting Event Delivery

11.3.1 IgnoreNewKeyboards

When Xkb is initialized, it implicitly forces requests for `NewKeyboardNotify` events. These events may be used by the Xkb library extension internally; they are normally translated into core protocol `MappingNotify` events before being passed to the client. While delivering the event to the client is appropriate in most cases, it is not appropriate for some clients which maintain per-key data structures. This is because once the server has sent a `NewKeyboardNotify` event, it is free to send the client events for all keys in the new range, and that range may be outside of the per-key data structures the client is maintaining.

The `IgnoreNewKeyboards` control, if enabled, prevents Xkb from mapping `NewKeyboardNotify` events to core `MappingNotify` events and passing them to the client. The control is initially disabled.

11.4 Manipulating the Library Controls

The Library Controls are manipulated using functions which deal with bitmasks to indicate which controls to manipulate. The controls are identified by the masks defined in Table 11.1

Table 11.1 Library Control Masks

<i>Library Control Mask</i>	<i>Value</i>
<code>XkbLC_ForceLatin1Lookup</code>	$(1 \ll 0)$
<code>XkbLC_ConsumeLookupMods</code>	$(1 \ll 1)$
<code>XkbLC_AlwaysConsumeShiftAndLock</code>	$(1 \ll 2)$
<code>XkbLC_IgnoreNewKeyboards</code>	$(1 \ll 3)$
<code>XkbLC_ConsumeKeysOnComposeFail</code>	$(1 \ll 29)$
<code>XkbLC_ComposeLED</code>	$(1 \ll 30)$
<code>XkbLC_BeepOnComposeFail</code>	$(1 \ll 31)$
<code>XkbLC_AllControls</code>	<code>(0xc0000007)</code>

11.4.1 Determining which Library Controls are Implemented

To determine which Library Controls are actually implemented, call `XkbXlibControlsImplemented`.

```
unsigned int XkbXlibControlsImplemented(display)
    Display *           display;          /* connection to X server */
```

`XkbXlibControlsImplemented` returns a bitmask indicating the controls actually implemented in the Xkb library, and is composed of an inclusive OR of bits from Table 11.1.

11.4.2 Determining the State of the Library Controls

To determine the current state of the Library Controls, call `XkbGetXlibControls`.

```
unsigned int XkbGetXlibControls(display)
    Display *           display;          /* connection to X server */
```

`XkbGetXlibControls` returns the current state of the Library Controls as a bit mask which is an inclusive OR of the control masks from Table 11.1 for the controls that are enabled. For the optional compose processing controls, the fact that a control is enabled does not imply that it is actually implemented.

11.4.3 Changing the State of the Library Controls

To change the state of the Library Controls, call `XkbSetXlibControls`.

```
Bool XkbSetXlibControls(display, bits_to_change, values_for_bits)
    Display *           display;          /* connection to X server */
    unsigned long        bits_to_change;  /* selects controls to be modified */
    unsigned long        values_for_bits; /* turns selected controls on (1) or off (0) */
```

XkbSetXlibControls modifies the state of the controls selected by *bits_to_change*; only the controls selected by *bits_to_change* are modified. If the bit corresponding to a control is on in *bits_to_change* and also on in *values_for_bits*, the control is enabled. If the bit corresponding to a control is on in *bits_to_change* but off in *values_for_bits*, the control is disabled. *bits_to_change* should be an inclusive OR of bits from Table 11.1.

12 Interpreting Key Events

Xkb provides functions to help developers interpret key events without having to directly interpret Xkb data structures. Xkb also modifies the behavior of several core X library functions.

12.1 Effects of Xkb on the Core X Library

When support for Xkb is built into the X library, the *XOpenDisplay* routine looks for a compatible version of Xkb on the server. If it finds a compatible version, it initializes the extension and enables *implicit support* for Xkb in a number of X library functions. This makes it possible for clients to take advantage of nearly all Xkb features without having to be rewritten or even recompiled, if they are built with shared libraries. This implicit support is invisible to most clients, but it can have side-effects, so the extension includes ways to control or disable it.

12.1.1 Effects of Xkb on Event State

Because *XOpenDisplay* initializes Xkb, some events contain an Xkb description of the keyboard state instead of that normally used by the core protocol. See section 17.1.1 for more information about the differences between Xkb keyboard state and that reported by the core protocol.

12.1.2 Effects of Xkb on MappingNotify Events

When Xkb is missing or disabled, the X library tracks changes to the keyboard mapping using *MappingNotify* events. Whenever the keyboard mapping is changed, the server sends all clients a *MappingNotify* event to report the change. When a client receives a *MappingNotify* event, it is supposed to call *XRefreshKeyboardMapping* to update the keyboard description used internally by the X library.

The X Keyboard Extension uses *XkbMapNotify* and *XkbNewKeyboardNotify* events to track changes to the keyboard mapping. When an Xkb-aware client receives either event, it should call *XkbRefreshKeyboardMapping* to update the keyboard description used internally by the X library. To avoid duplicate events, the X server does not send core protocol *MappingNotify* events to a client that has selected for *XkbMapNotify* events.

The implicit support for Xkb selects for *XkbMapNotify* events. This means that clients that do not explicitly use Xkb but which are using a version of the X library that has implicit support for Xkb do not receive *MappingNotify* events over the wire. Clients that were not written with Xkb in mind do not recognize or properly handle the new Xkb events, so the implicit support converts them to *MappingNotify* events which report approximately the same information, unless the client has explicitly selected for the Xkb version of the event.

An Xkb-capable X server does not send events from keys that fall outside the legal range of keycodes expected by that client. Once the server sends a client an *XkbNewKeyboardNotify* event, it reports events from all keys because it assumes that any client which has received an *XkbNewKeyboardNotify* event expects key events from the new range of keycodes. The implicit support for Xkb asks for *XkbNewKeyboardNotify* events, so the range of keycodes reported to the client might vary without the client's knowledge. Most clients don't really care about the range of legal keycodes, but some clients maintain information about each key and might have problems with events that come from unex-

pected keys. Such clients can set the `XkbLC_IgnoreNewKeyboards` library control (see section 11.3.1 on page 84) to prevent the implicit support from requesting notification of changes to the legal range of keycodes.

12.1.3 X Library Functions Affected by Xkb

The following X library functions are modified by Xkb:

XKeyCodeToKeysym
XKeysymToKeyCode
XLookupKeysym
XLookupString
XRefreshKeyboardMapping
XRebindKeysym

The implicit support for Xkb replaces a number of X library functions with versions that understand and use the X Keyboard Extension. In most cases, the semantics of the new versions are identical to those of the old, but there are occasional visible differences. This section lists all of the functions that are affected, and the differences in behavior, if any, that are visible to clients.

The *XKeyCodeToKeysym* function reports the keysym associated with a particular index for a single key. The index specifies a column of symbols in the core keyboard mapping (that is, as reported by the core protocol *GetKeyboardMapping* request). The order of the symbols in the core mapping does not necessarily correspond to the order of the symbols used by Xkb; section 17.1.3 describes the differences.

The *XKeysymToKeyCode* function reports a keycode to which a particular keysym is bound. When Xkb is missing or disabled, this function looks in each column of the core keyboard mapping in turn, and returns the lowest numbered key that matches in the lowest numbered group. When Xkb is present, this function uses the Xkb ordering for symbols, instead.

The *XLookupKeysym* function reports the symbol in a specific column of the key associated with an event. Whether or not Xkb is present, the column specifies an index into the core symbol mapping.

The *XLookupString* function reports the symbol and string associated with a key event, taking into account the keycode and keyboard state as reported in the event. When Xkb is disabled or missing, *XLookupString* uses the rules specified by the core protocol and reports only ISO Latin-1 characters. When Xkb is present, *XLookupString* uses the explicit keyboard group, key types and rules specified by Xkb. When Xkb is present, *XLookupString* is allowed, but not required, to return strings in character sets other than ISO Latin-1, depending on the current locale. If any key bindings are defined, *XLookupString* does not use any consumed modifiers (see section 11.1.2 and section 15.2) to determine matching bindings.

The *XRefreshKeyboardMapping* function updates the X library's internal representation of the keyboard to reflect changes reported via *MappingNotify* events. When Xkb is missing or disabled, this function reloads the entire modifier map or keyboard mapping. When Xkb is present, the implicit Xkb support keeps track of the changed components reported by each *XkbMapNotify* event and updates only those pieces of the keyboard description that have changed. If the implicit support has not noted any keyboard mapping changes, *XRefreshKeyboardMapping* updates the entire keyboard description.

The *XRebindKeysym* function associates a string with a keysym and a set of modifiers. Xkb does not directly change this function, but it does affect the way that the state reported in the event is compared to the state specified to *XRebindKeysym*. When Xkb is missing or disabled, *XLookupString* returns the specified string if the modifiers in the event exactly match the modifiers from this call. When Xkb is present, any modifiers used to determine the keysym are consumed and are not used to look up the string.

12.2 Xkb Event and Keymap Functions

To find the keysym bound to a particular key at a specified group and shift level, call *XkbKeyCodeToKeysym*.

```
KeySym XkbKeyCodeToKeysym(dpy, kc, group, level)
    Display *          dpy;           /* connection to X server */
    KeyCode            kc;           /* key of interest */
    unsigned int       group;        /* group of interest */
    unsigned int       level;        /* shift level of interest */
```

XkbKeyCodeToKeysym returns the keysym bound to a particular group and shift level for a particular key on the core keyboard. If *kc* is not a legal keycode for the core keyboard, or if *group* or *level* are out of range for the specified key, *XkbKeyCodeToKeysym* returns *NoSymbol*.

To find the set of modifiers bound to a particular keysym on the core keyboard, call *XkbKeysymToModifiers*.

```
unsigned int XkbKeysymToModifiers(dpy, ks)
    Display *          dpy;           /* connection to X server */
    KeySym             ks;           /* keysym of interest */
```

XkbKeysymToModifiers finds the set of modifiers currently bound to the keysym *ks* on the core keyboard. The value returned is the mask of modifiers bound to the keysym *ks*. If no modifiers are bound to the keysym, *XkbKeysymToModifiers* returns zero; otherwise it returns the inclusive OR of zero or more of the following: ShiftMask, ControlMask, LockMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask and Mod5Mask.

Call *XkbLookupKeySym* to find the symbol associated with a key for a particular state.

```
Bool XkbLookupKeySym(dpy, key, state, mods_rtrn, sym_rtrn)
    Display *          dpy;           /* connection to X server */
    KeyCode            key;           /* key for which symbols are to be found */
    unsigned int       state;         /* state for which symbol should be found */
    unsigned int *     mods_rtrn;      /* backfilled with unconsumed modifiers */
    KeySym *           sym_rtrn;      /* backfilled with symbol associated with key + state */
```

XkbLookupKeySym is the equivalent of the core *XLookupKeySym* function. For the core keyboard, given a keycode *key* and an Xkb state *state*, *XkbLookupKeySym* returns the symbol associated with the key in *sym_rtrn* and the list of modifiers that should still be applied in *mods_rtrn*. The *state* parameter is the state from a KeyPress or KeyRelease event. *XkbLookupKeySym* returns True if it succeeds.

Call *XkbLookupKeyBinding* to find the string bound to a key by *XRebindKeySym*. *XkbLookupKeyBinding* is the equivalent of the core *XLookupString* function.

```
int XkbLookupKeyBinding(dpy, sym, state, buf, nbytes, extra_rtrn)
    Display *          dpy;           /* connection to server */
    KeySym            sym;           /* symbol to be looked up */
    unsigned int       state;          /* state for which string is to be looked up */
    char *             buf;           /* buffer into which returned string is written */
    int                nbytes;         /* size of buffer in bytes */
    int *              extra_rtrn;     /* backfilled with number bytes overflow */
```

XRebindKeysym binds an ASCII string to a specified keysym, so that the string and keysym are returned when the key is pressed and a specified list of modifiers are also being held down. *XkbLookupKeyBinding* returns in *buf* the string associated with the keysym *sym* and modifier state *state*. *buf* is NULL terminated unless there's an overflow. If the string returned is larger than *nbytes*, a count of bytes that does not fit into the buffer is returned in *extra_rtrn*. *XkbTranslateKeySym* returns the number of bytes that it placed into *buf*.

To find the string and symbol associated with a keysym for a given keyboard state, call *XkbTranslateKeySym*.

```
int XkbTranslateKeySym(dpy, sym_inout, mods, buf, nbytes, extra_rtrn)
    Display *          dpy;           /* connection to X server */
    KeySym *           sym_inout;      /* symbol to be translated; result of translation */
    unsigned int        mods;          /* modifiers to apply to sym_inout */
    char *              buf;           /* buffer into which returned string is written */
    int                 nbytes;         /* size of buffer in bytes */
    int *              extra_rtrn;     /* number of bytes overflow*/
```

XkbTranslateKeySym applies the transformations specified in *mods* to the symbol specified by *sym_inout*. It returns in *buf* the string, if any, associated with the keysym for the current locale. If the transformations in *mods* changes the keysym, *sym_inout* is updated accordingly. If the string returned is larger than *nbytes*, a count of bytes that does not fit into the buffer is returned in *extra_rtrn*. *XkbTranslateKeySym* returns the number of bytes it placed into *buf*.

To update the keyboard description that is internal to the X library, call *XkbRefreshKeyboardMapping*.

```
Status XkbRefreshKeyboardMapping(event)
    XkbMapNotifyEvent * event;        /* event initiating remapping */
```

XkbRefreshKeyboardMapping is the Xkb equivalent of the core *XRefreshKeyboardMapping* function. It requests that the X server send the current key mapping information to this client. A client usually invokes *XkbRefreshKeyboardMapping* after receiving an *XkbMapNotify* event. *XkbRefreshKeyboardMapping* returns Success if it succeeds and BadMatch if the event is not an Xkb event.

The *XkbMapNotify* event can be generated when some client calls *XkbSetMap*, *XkbChangeMap*, *XkbGetKeyboardByName*, or any of the standard X library functions that change the keyboard mapping or modifier mapping.

To translate a keycode to a key symbol and modifiers, call *XkbTranslateKeyCode*.

```
Booll XkbTranslateKeyCode(xkb, key, mods, mods_rtrn, keysym_rtrn)
    XkbDescPtr      xkb;          /* keyboard description to use for translation */
    KeyCode         key;          /* keycode to translate */
    unsigned int    mods;         /* modifiers to apply when translating key */
    unsigned int *  mods_rtrn;   /* backfilled with unconsumed modifiers */
    KeySym *        keysym_rtrn; /* keysym resulting from translation */
```

mods_rtrn is backfilled with the modifiers consumed by the translation process. *mods* is a bit-wise inclusive OR of the legal modifier masks: ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, Mod5Mask. The AlwaysConsume-ShiftAndLock library control (see section 11.1.3), if enabled, causes *XkbTranslateKeyCode* to consume shift and lock. *XkbTranslateKeyCode* returns True if the translation resulted in a keysym, and False if it resulted in NoSymbol.

13 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.

You can retrieve a keyboard geometry from an X server that supports Xkb, or you can allocate it from scratch and initialize it in a client program. The keyboard geometry need not have any correspondence with the physical keyboard that is connected to the X server.

Geometry measurements are specified in $\text{mm}/10$ units. The origin (0,0) is in the top left corner of the keyboard image. A component's own origin is also its upper left corner. In some cases a component needs to be drawn rotated. For example, a special keyboard may have a section of keys arranged in rows in a rectangular area, but the entire rectangle may not be in alignment with the rest of the keyboard, and instead, it is rotated from horizontal by 30° . Rotation for a geometry object is specified in $1/10^\circ$ increments about its origin. An example of a keyboard with rotated sections is shown in Figure 131.

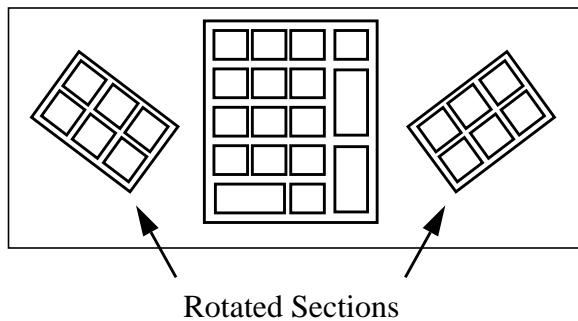


Figure 131 Rotated Keyboard Sections

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

The keyboard geometry's top-level description is stored in a `XkbGeometryRec` structure. This structure contains three types of information:

1. Lists of items, not used to draw the basic keyboard, but indexed by the geometry descriptions which comprise the entire keyboard geometry (colors, geometry properties, key aliases, shapes)
2. A number of singleton items which describe the keyboard as a whole (keyboard name, width and height, a color for the keyboard as a whole, and a color for keyboard key labels)
3. A list of the keyboard's sections and non-key doodads

The top-level geometry is described in more detail below.

The lists of items used by components of the keyboard geometry description is as follows:

- The top-level keyboard geometry description includes a list of up to `MaxColors` (32) *color names*. A color name is a string whose interpretation is not specified by Xkb. The `XkbColorRec` structure provides a field for this name as well as a pixel field. The pixel field is a convenient place for an application to store a pixel value or color definition, if it needs to. All other geometry data structures refer to colors using their indices in this global list.
- The top-level keyboard geometry description includes 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. As an example of a possible use of *properties*, consider the pause/break key on most PC keyboards: the “break” symbol is usually on the front of the key, and is often a different color. A program might set a property to:
`LBL_PAUS = “{Pause/top/black,Break/front/red}”`
and use the property information to draw the key with a front label as well as a top label.
- The top-level keyboard geometry description includes a list of *key aliases* (see Chapter 18). Key aliases allow the keyboard layout designer to assign multiple key names to a single key.

Note Key aliases defined in the geometry component of a keyboard mapping override those defined in the keycodes component of the server database, which are stored in the `XkbNamesRec` (`xb->names`). Therefore, consider the key aliases defined by the geometry before considering key aliases supplied by the keycodes.

- The top-level keyboard geometry description includes a list of *shapes*; other keyboard components refer to shapes by their index in this list. A shape consists of an arbitrary name of type Atom and one or more closed-polygon *outlines*. All points in an outline are specified relative to the origin of its enclosing shape, that is, whichever shape that contains this outline in its list of outlines. One outline is the primary outline. The primary outline is by default the first outline, or it can be optionally specified by the *primary* field in the `XkbShapeRec` structure. 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.

The `XkbGeometryRec` top-level geometry description contains the following information which pertains to the keyboard as a whole:

- A *keyboard symbolic name* of type Atom to help users identify the keyboard.
- The *width* and *height* of the keyboard, in $\text{mm}/10$. For non-rectangular keyboards, the width and height describe the smallest bounding-box that encloses the outline of the keyboard.
- 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; label fonts are arbitrary strings, since Xkb does not specify format or name space for font names.

The keyboard is subdivided into named *sections* of related keys and *doodads*. The sections and doodads on the keyboard are listed in the `XkbGeometryRec` top-level keyboard geometry description. A *section* is composed of keys that are physically together and logi-

cally related. Figure 13.2 shows a keyboard that is divided into four sections. A *doodad* describes some visible aspect of the keyboard that is not a key and is not a section.

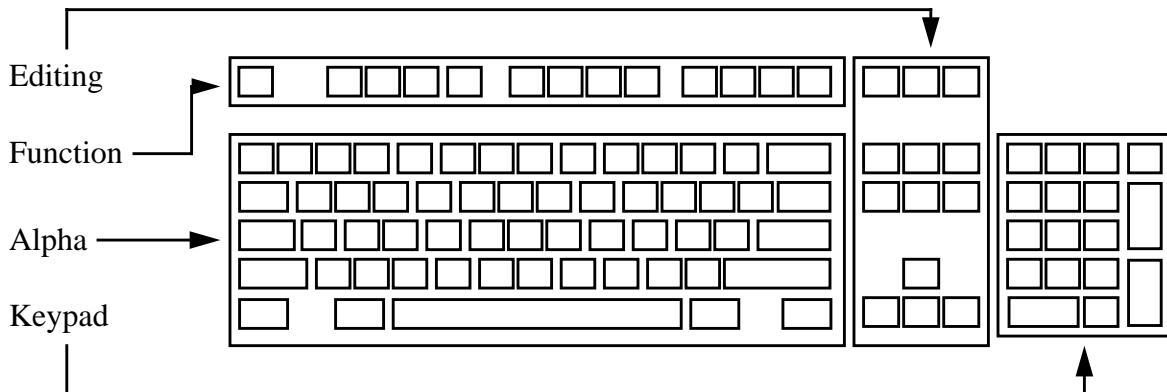


Figure 13.2 Keyboard with Four Sections

13.1 Shapes and Outlines

A *shape*, used to draw keyboard components and stored in a `XkbShapeRec` structure, has:

- An arbitrary name of type Atom.
- Bounds (two x and y coordinates) that describe the corners of a rectangle containing the shape's top surface outline.
- A list of one or more outlines (described below).
- Optional pointers to a primary and an approximation outline (described below). If either of these pointers is `NULL`, the default primary/approximation outline is the first one in the list of outlines for the shape.

An *outline*, stored in a `XkbOutlineRec` structure, 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 *corner_radius* 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 is the primary outline; a keyboard display application can generate a simple but still accurate keyboard image by displaying only the primary outlines for each shape. The default primary outline is the first in a shape's list of outlines. If the *primary* field of the `XkbShapeRec` structure is not `NULL`, it points to the primary outline. A rectangular *approximation* must be included for non-rectangular keys 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.

13.2 Sections

As noted above, a keyboard is subdivided into *sections* of related keys. 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, stored in a `XkbSectionRec`, include:

- An arbitrary name of type Atom.
- A priority, to indicate drawing order. 0 is the highest priority, 255 the lowest.
- Origin of the section, relative to the origin of the keyboard.
- The width and height, and the angle of rotation.
- 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. Figure 13.3 shows the alpha section from the keyboard shown in Figure 13.2, divided into rows. Rows and keys are defined below.

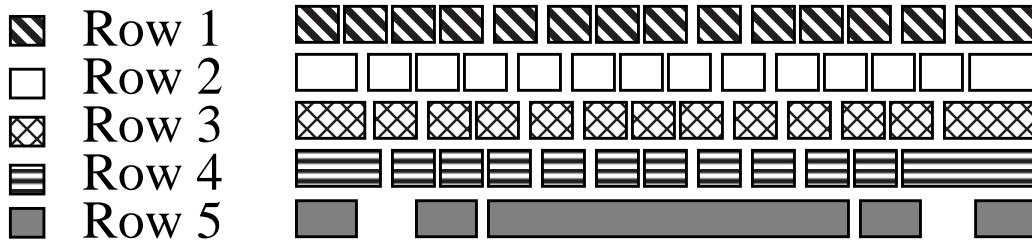


Figure 13.3 Rows in a Section

- 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 for doodads in a section is relative to the other components of the section, not to the keyboard as a whole.
- An optional *overlay* with a name of type Atom and a list of overlay rows (described below).
- Bounds (two x and y coordinates) that describe the corners of a rectangle containing the entire section.

13.3 Rows and Keys

A row description (`XkbRowRec`) consists of the coordinates of its origin relative to its enclosing section, a flag indicating whether the row is horizontal or vertical, and a list of keys in the row.

A key description (`XkbKeyRec`) 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; the label color is stored in the `XkbGeometryRec`). 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.

13.4 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, if present, describes the outline of the entire keyboard.

Each doodad’s origin is stored in fields named *left* and *top*, which are the coordinates of the doodad’s origin relative to its enclosing object, whether it be a section or the top-level keyboard. The priority for doodads which are listed in the top-level geometry are relative to the other doodads listed in the top-level geometry and the sections listed in the top-level geometry. The priority for doodads which are listed in a section are relative to the other components of the section. Each doodad is stored in a structure with a *type* field, which specifies the type of doodad.

Xkb supports five types 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.

The structures these doodads are stored in and the values of the *type* fields are shown in Table 13.1.

Table 13.1 Doodad Types

Doodad	Structure	Type
<i>indicator doodad</i>	XkbIndicatorDoodadRec	XkbIndicatorDoodad
<i>outline doodad</i>	XkbShapeDoodadRec	XkbOutlineDoodad
<i>solid doodad</i>	XkbShapeDoodadRec	XkbSolidDoodad
<i>text doodad</i>	XkbTextDoodadRec	XkbTextDoodad
<i>logo doodad</i>	XkbLogoDoodadRec	XkbLogoDoodad

13.5 Overlay Rows and Overlay Keys

An *overlay row* (XkbOverlayRowRec) contains a pointer to the row it overlays, and a list of *overlay keys*.

Each overlay key definition (XkbOverlayKeyRec) indicates a key that can yield multiple keycodes 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 key-code 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 be.

13.6 Drawing a Keyboard Representation

To draw a representation of the keyboard, draw in the following order:

Draw the top-level keyboard as a rectangle, using its width and height.

For each component (section or doodad) of the top-level geometry, in priority order:

If component is a section

 For each row, in the order it appears in the section

 Draw keys in the order they appear in the row

 Draw doodads within the section in priority order.

Else draw doodad

13.7 Geometry Data Structures

In the following figures, a solid arrow denotes a pointer to an array of structures or a singleton structure. A dotted arrow denotes an index or a pointer into the array.

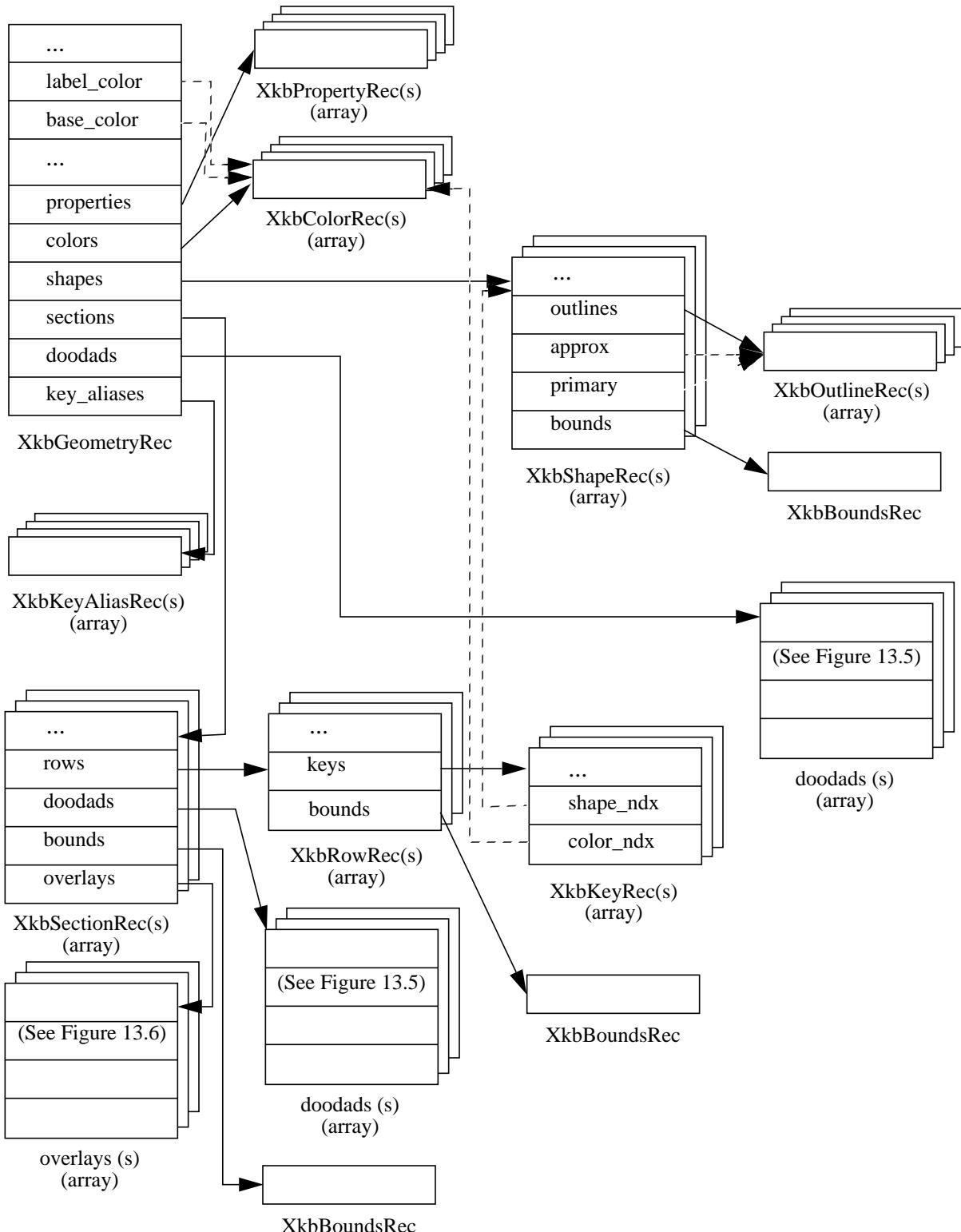
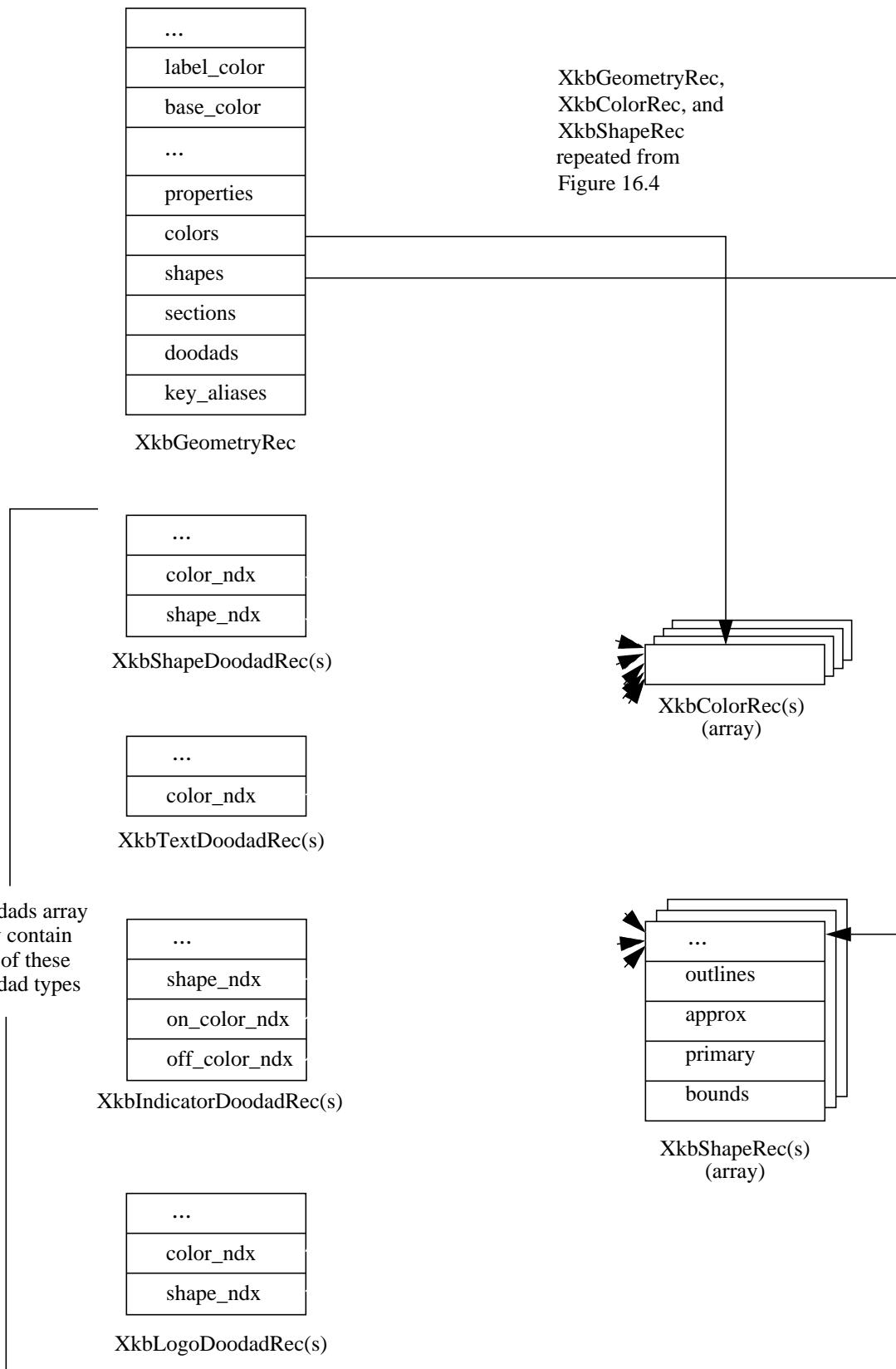
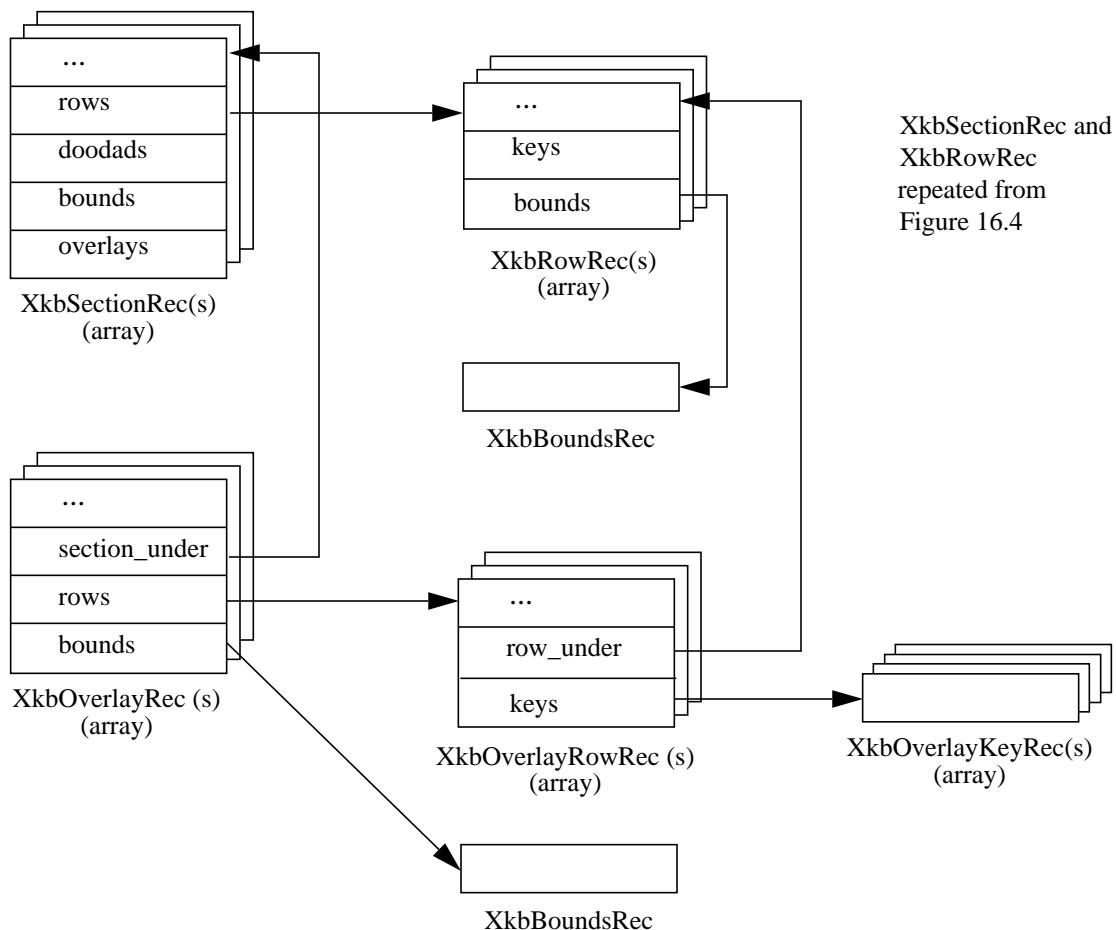


Figure 13.4 Xkb Geometry Data Structures

**Figure 13.5 Xkb Geometry Data Structures (Doodads)**

**Figure 13.6 Xkb Geometry Data Structures (Overlays)**

```

typedef struct _XkbGeometry {
    /* top-level keyboard geometry structure */
    Atom           name;          /* keyboard name */
    unsigned short width_mm;     /* keyboard width in mm/10 */
    unsigned short height_mm;    /* keyboard height in mm/10 */
    char *         label_font;   /* font for key labels */
    XkbColorPtr   label_color;  /* color for key labels - pointer into colors array */
    XkbColorPtr   base_color;   /* color for basic keyboard - pointer into colors array */
    unsigned short sz_properties; /* size of properties array */
    unsigned short sz_colors;   /* size of colors array */
    unsigned short sz_shapes;   /* size of shapes array */
    unsigned short sz_sections; /* size of sections array */
    unsigned short sz_doodads;  /* size of doodads array */
    unsigned short sz_key_aliases; /* size of key aliases array */
    num_properties; /* number of properties in the properties array */
    num_colors;    /* number of colors in the colors array */
    num_shapes;   /* number of shapes in the shapes array */
    num_sections; /* number of sections in the sections array */
    num_doodads;  /* number of doodads in the doodads array */
}

```

```

    unsigned short num_key_aliases; /* number of key aliases in the key */
    XkbPropertyPtr properties; /* properties array */
    XkbColorPtr colors; /* colors array */
    XkbShapePtr shapes; /* shapes array */
    XkbSectionPtr sections; /* sections array */
    XkbDoodadPtr doodads; /* doodads array */
    XkbKeyAliasPtr key_aliases; /* key aliases array */
} XkbGeometryRec *XkbGeometryPtr;

```

The *doodads* array is only for doodads not contained in any of the *sections* which has its own *doodads*. The key aliases contained in the *key_aliases* array take precedence over any defined in the keycodes component of the keyboard description.

```

typedef struct _XkbProperty {
    char * name; /* property name */
    char * value; /* property value */
} XkbPropertyRec, *XkbPropertyPtr;

typedef struct _XkbColor {
    unsigned int pixel; /* color */
    char * spec; /* color name */
} XkbColorRec, *XkbColorPtr;

typedef struct _XkbKeyAliasRec {
    char real[XkbKeyNameLength]; /* real name of the key */
    char alias[XkbKeyNameLength]; /* alias for the key */
} XkbKeyAliasRec, *XkbKeyAliasPtr;

typedef struct _XkbPoint { /* x,y coordinates */
    short x;
    short y;
} XkbPointRec, *XkbPointPtr;

typedef struct _XkbOutline {
    unsigned short num_points; /* number of points in the outline */
    unsigned short sz_points; /* size of the points array */
    unsigned short corner_radius; /* draw corners as circles with this radius */
    XkbPointPtr points; /* array of points defining the outline */
} XkbOutlineRec, *XkbOutlinePtr;

typedef struct _XkbBounds {
    short x1,y1; /* upper left corner of the bounds, in mm/10 */
    short x2,y2; /* lower right corner of the bounds, in mm/10 */
} XkbBoundsRec, *XkbBoundsPtr;

typedef struct _XkbShape {
    Atom name; /* shape's name */
    unsigned short num_outlines; /* number of outlines for the shape */
    unsigned short sz_outlines; /* size of the outlines array */
    XkbOutlinePtr outlines; /* array of outlines for the shape */
    XkbOutlinePtr approx; /* pointer into the array to the approximating outline */
    XkbOutlinePtr primary; /* pointer into the array to the primary outline */
    XkbBoundsRec bounds; /* bounding box for the shape; encompasses all outlines */
} XkbShapeRec, *XkbShapePtr;

```

If *approx* and/or *primary* is NULL, the default value is used. The default primary outline is the first element in the outlines array, as is the default approximating outline.

```
typedef struct _XkbKey {           /* key in a row */
    XkbKeyNameRec  name;          /* key name */
    short           gap;           /* gap in mm/10 from previous key in row */
    unsigned char   shape_ndx;     /* index of shape for key */
    unsigned char   color_ndx;    /* index of color for key body */
} XkbKeyRec, *XkbKeyPtr;
```

```
typedef struct _XkbRow {           /* row in a section */
    short           top;           /* top coordinate of row origin, relative to section's origin */
    short           left;          /* left coordinate of row origin, relative to section's origin */
    unsigned short  num_keys;      /* number of keys in the keys array */
    unsigned short  sz_keys;       /* size of the keys array */
    int             vertical;     /* True=>vertical row, False=>horizontal row */
    XkbKeyPtr       keys;          /* array of keys in the row*/
    XkbBoundsRec   bounds;        /* bounding box for the row */
} XkbRowRec, *XkbRowPtr;
```

top and *left* are in mm/10.

```
typedef struct _XkbOverlayRec {
    Atom            name;          /* overlay name */
    XkbSectionPtr   section_under; /* the section under this overlay */
    unsigned short  num_rows;     /* number of rows in the rows array */
    unsigned short  sz_rows;      /* size of the rows array */
    XkbOverlayRowPtr rows;        /* array of rows in the overlay */
    XkbBoundsPtr    bounds;        /* bounding box for the overlay */
} XkbOverlayRec, *XkbOverlayPtr;
```

```
typedef struct _XkbOverlayRow {
    unsigned short  row_under;    /* index into the row under this overlay row */
    unsigned short  num_keys;     /* number of keys in the keys array */
    unsigned short  sz_keys;      /* size of the keys array */
    XkbOverlayKeyPtr keys;        /* array of keys in the overlay row */
} XkbOverlayRowRec, *XkbOverlayRowPtr;
```

row_under is an index into the array of *rows* in the section under this overlay. The section under this overlay row is the one pointed to by *section_under* in this overlay row's *XkbOverlayRec*.

```
typedef struct _XkbOverlayKey {
    XkbKeyNameRec  over;         /* name of this overlay key */
    XkbKeyNameRec  under;        /* name of the key under this overlay key */
} XkbOverlayKeyRec, *XkbOverlayKeyPtr;
```

```
typedef struct _XkbSection {
    Atom            name;          /* section name */
    unsigned char   priority;      /* drawing priority, 0=>highest, 255=>lowest */
    short           top;           /* top coordinate of section origin */
    short           left;          /* left coordinate of row origin */
    unsigned short  width;         /* section width, in mm/10 */
    unsigned short  height;        /* section height, in mm/10 */
} XkbSectionRec, *XkbSectionPtr;
```

```

short           angle;      /* angle of section rotation, counterclockwise */
unsigned short num_rows;   /* number of rows in the rows array */
unsigned short num_doodads; /* number of doodads in the doodads array */
unsigned short num_overlays; /* number of overlays in the overlays array */
unsigned short sz_rows;    /* size of the rows array */
unsigned short sz_doodads; /* size of the doodads array */
unsigned short sz_overlays; /* size of the overlays array */
XkbRowPtr      rows;       /* section rows array */
XkbDoodadPtr   doodads;   /* section doodads array */
XkbBoundsRec   bounds;    /* bounding box for the section, before rotation*/
XkbOverlayPtr  overlays;   /* section overlays array */

} XkbSectionRec, *XkbSectionPtr;

```

top and *left* are the origin of the section, relative to the origin of the keyboard, in mm/10. *angle* is in $1/10$ degrees.

DoodadRec Structures

The doodad arrays in the XkbGeometryRec and the XkbSectionRec may contain any of the doodad structures and types shown in Table 13.1 on page 96.

The doodad structures form a union:

```

typedef union _XkbDoodad {
    XkbAnyDoodadRec      any;
    XkbShapeDoodadRec    shape;
    XkbTextDoodadRec     text;
    XkbIndicatorDoodadRec indicator;
    XkbLogoDoodadRec    logo;
} XkbDoodadRec, *XkbDoodadPtr;

```

The *top* and *left* coordinates of each doodad are the coordinates of the origin of the doodad relative to the keyboard's origin if the doodad is in the XkbGeometryRec doodad array, and with respect to the section's origin if the doodad is in a XkbSectionRec doodad array. The *color_ndx* or *on_color_ndx* and *off_color_ndx* fields are color indices into the XkbGeometryRec's color array, and are the colors to draw the doodads with. Similarly, the *shape_ndx* fields are indices into the XkbGeometryRec's shape array.

```

typedef struct _XkbShapeDoodad {
    Atom          name;      /* doodad name */
    unsigned char type;     /* XkbOutlineDoodad or XkbSolidDoodad */
    unsigned char priority; /* drawing priority, 0=>highest, 255=>lowest */
    short         top;       /* top coordinate, in mm/10 */
    short         left;      /* left coordinate, in mm/10 */
    short         angle;     /* angle of rotation, clockwise, in 1/10 degrees */
    unsigned short color_ndx; /* doodad color */
    unsigned short shape_ndx; /* doodad shape */
} XkbShapeDoodadRec, *XkbShapeDoodadPtr;

```

```

typedef struct _XkbTextDoodad {
    Atom          name;      /* doodad name */
    unsigned char type;     /* XkbTextDoodad */
    unsigned char priority; /* drawing priority, 0=>highest, 255=>lowest */
    short         top;       /* top coordinate, in mm/10 */
}

```

```

short      left;          /* left coordinate, in mm/10 */
short      angle;         /* angle of rotation, clockwise, in 1/10 degrees */
short      width;         /* width in mm/10 */
short      height;        /* height in mm/10 */
unsigned short color_ndx; /* doodad color */
char *     text;          /* doodad text */
char *     font;          /* arbitrary font name for doodad text */
} XkbTextDoodadRec, *XkbTextDoodadPtr;

typedef struct _XkbIndicatorDoodad {
    Atom      name;          /* doodad name */
    unsigned char type;        /* XkbIndicatorDoodad */
    unsigned char priority;   /* drawing priority, 0=>highest, 255=>lowest */
    short      top;           /* top coordinate, in mm/10 */
    short      left;          /* left coordinate, in mm/10 */
    short      angle;         /* angle of rotation, clockwise, in 1/10 degrees */
    unsigned short shape_ndx; /* doodad shape */
    unsigned short on_color_ndx; /* color for doodad if indicator is on */
    unsigned short off_color_ndx; /* color for doodad if indicator is off */
} XkbIndicatorDoodadRec, *XkbIndicatorDoodadPtr;

typedef struct _XkbLogoDoodad {
    Atom      name;          /* doodad name */
    unsigned char type;        /* XkbLogoDoodad */
    unsigned char priority;   /* drawing priority, 0=>highest, 255=>lowest */
    short      top;           /* top coordinate, in mm/10 */
    short      left;          /* left coordinate, in mm/10 */
    short      angle;         /* angle of rotation, clockwise, in 1/10 degrees */
    unsigned short color_ndx; /* doodad color */
    unsigned short shape_ndx; /* doodad shape */
    char *     logo_name;     /* text for logo */
} XkbLogoDoodadRec, *XkbLogoDoodadPtr

```

13.8 Getting Keyboard Geometry From the Server

You can load a keyboard geometry as part of the keyboard description returned by *XkbGetKeyboard*. However, if a keyboard description has been previously loaded, you can instead obtain the geometry by calling the *XkbGetGeometry*. In this case, the geometry returned is the one associated with the keyboard whose device id is contained in the keyboard description.

To load a keyboard geometry if you already have the keyboard description, call *XkbGetGeometry*.

Status **XkbGetGeometry**(*dpy*, *xkb*)

Display *	<i>dpy</i> ;	/* connection to the X server */
XkbDescPtr	<i>xkb</i> ;	/* keyboard description that contains the id for the keyboard and into which the geometry should be loaded */

XkbGetGeometry can return *BadValue*, *BadImplementation*, *BadName*, *BadAlloc* or *BadLength* errors, or *Success* if it succeeds.

It is also possible to load a keyboard geometry by name. The X server maintains a database of keyboard components (see Chapter 20). To load a keyboard geometry description from this database by name, call *XkbGetNamedGeometry*.

```
Status XkbGetNamedGeometry(dpy, xkb, name)
    Display *      dpy;      /* connection to the X server */
    XkbDescPtr   xkb;      /* keyboard description into which the geometry should be loaded */
    Atom         name;      /* name of the geometry to be loaded */
```

XkbGetNamedGeometry can return *BadName* if the *name* cannot be found.

13.9 Using Keyboard Geometry

Xkb provides a number of convenience functions to help use a keyboard geometry. These include functions to return the bounding box of a shape's top surface, and to update the bounding box of a shape row or section.

A shape is made up of a number of outlines. Each outline is a polygon made up of a number of points. The bounding box of a shape is a rectangle that contains all the outlines of that shape.

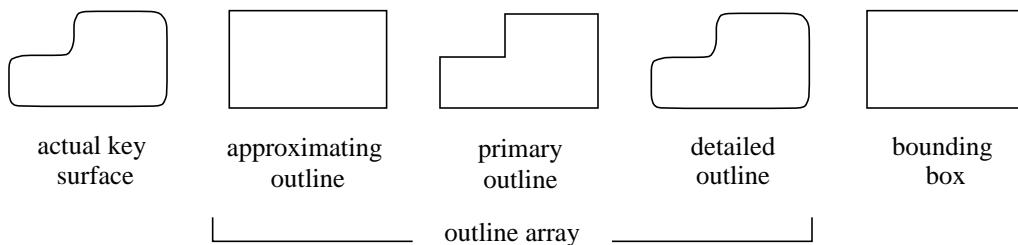


Figure 13.7 Key Surface, Shape Outlines and Bounding Box

To determine the bounding box of the top surface of a shape, call *XkbComputeShapeTop*.

```
Bool XkbComputeShapeTop(shape, bounds_rtrn)
    XkbShapePtr   shape;      /* shape to be examined */
    XkbBoundsPtr  bounds_rtrn /* backfilled with the bounding box */
```

XkbComputeShapeTop returns a *BoundsRec* that contains two x and y coordinates. These coordinates describe the corners of a rectangle that contains the outline that describes the top surface of the shape. The top surface is defined to be the approximating outline if the *approx* field of *shape* is not *NULL*. If *approx* is *NULL*, the top surface is defined to be the last outline in the *shape*'s array of outlines. *XkbComputeShapeTop* returns *False* if *shape* is *NULL* or if there are no outlines for the shape; otherwise it returns *True*.

A *ShapeRec* contains a *BoundsRec* that describes the bounds of the shape. If you add or delete an outline to or from a shape, the bounding box must be updated. To update the bounding box of a shape, call *XkbComputeShapeBounds*.

```
Bool XkbComputeShapeBounds(shape)
    XkbShapePtr   shape;      /* shape to be examined */
```

XkbComputeShapeBounds updates the *BoundsRec* contained in the *shape* by examining all the outlines of the shape and setting the *BoundsRec* to the minimum x and minimum

y, and maximum *x* and maximum *y* values found in those outlines. *XkbComputeShapeBounds* returns *False* if *shape* is *NULL* or if there are no outlines for the shape; otherwise it returns *True*.

If you add or delete a key to or from a row, or if you update the shape of one of the keys in that row, you may need to update the bounding box of that row. To update the bounding box of a row, call *XkbComputeRowBounds*.

```
Bool XkbComputeRowBounds(geom, section, row)
    XkbGeometryPtr geom;          /* geometry which contains the section */
    XkbSectionPtr   section;       /* section which contains the row */
    XkbRowPtr      row;           /* row to be examined and updated */
```

XkbComputeRowBounds checks the bounds of all keys in the *row*, and updates the bounding box of the row if necessary. *XkbComputeRowBounds* returns *False* if any of the arguments is *NULL*; otherwise it returns *True*.

If you add or delete a row to or from a section, or if you change the geometry of any of the rows in that section, you may need to update the bounding box for that section. To update the bounding box of a section, call *XkbComputeSectionBounds*.

```
Bool XkbComputeSectionBounds(geom, section)
    XkbGeometryPtr geom;          /* geometry which contains the section */
    XkbSectionPtr   section;       /* section to be examined and updated */
```

XkbComputeSectionBounds examines all the rows of the *section* and updates the bounding box of that section so that it contains all rows. *XkbComputeSectionBounds* returns *False* if any of the arguments is *NULL*; otherwise it returns *True*.

Keys that can generate multiple keycodes may be associated with multiple names. Such keys have a primary name and an alternate name. To find the alternate name by using the primary name for a key that is part of an overlay, call *XkbFindOverlayForKey*.

```
char * XkbFindOverlayForKey(geom, section, under)
    XkbGeometryPtr geom;          /* geometry which contains the section */
    XkbSectionPtr   section;       /* section to be searched for matching keys */
    char *         under;        /* primary name of the key to be considered */
```

XkbFindOverlayForKey uses the primary name of the key, *under*, to look up the alternate name, which it returns.

13.10 Adding Elements to a Keyboard Geometry

Xkb provides functions to add a single new element to the top-level keyboard geometry. In each case the *num_** fields of the corresponding structure is incremented by 1. These functions do not change *sz_** unless there is no more room in the array. Some of these functions fill in the values of the element's structure from the arguments. For other functions, you must explicitly write code to fill the structure's elements.

The top-level geometry description includes a list of *geometry properties*. A geometry property associates an arbitrary string with an equally arbitrary name. Programs that display images of keyboards can use geometry properties as hints, but they are not interpreted by Xkb. No other geometry structures refer to geometry properties.

To add one property to an existing keyboard geometry description, call *XkbAddGeomProperty*.

```
XkbPropertyPtr XkbAddGeomProperty(geom, name, value)
    XkbGeometryPtr geom;          /* geometry to be updated */
    char *        name;          /* name of the new property */
    char *        value;          /* value for the new property */
```

XkbAddGeomProperty adds one property with the specified *name* and *value* to the keyboard geometry specified by *geom*. *XkbAddGeomProperty* returns NULL if any of the parameters is empty or if it was not able to allocate space for the property. To allocate space for an arbitrary number of properties, call the *XkbAllocGeomProps* function.

To add one key alias to an existing keyboard geometry description, call *XkbAddGeomKeyAlias*.

```
XkbKeyAliasPtr XkbAddGeomKeyAlias(geom, alias, real)
    XkbGeometryPtr geom;          /* geometry to be updated */
    char *        alias;          /* alias to be added */
    char *        real;           /* real name to be bound to the new alias */
```

XkbAddGeomKeyAlias adds one key alias with the value *alias* to the geometry *geom*, and associates it with the key whose real name is *real*. *XkbAddGeomKeyAlias* returns NULL if any of the parameters is empty or if it was not able to allocate space for the alias. To allocate space for an arbitrary number of aliases, call the *XkbAllocGeomKeyAliases* function.

To add one color name to an existing keyboard geometry description, call *XkbAddGeomColor*.

```
XkbColorPtr XkbAddGeomColor(geom, spec, pixel)
    XkbGeometryPtr geom;          /* geometry to be updated */
    char *        spec;           /* color to be added */
    unsigned int   pixel;          /* color to be added */
```

XkbAddGeomColor adds the specified color *name* and *pixel* to the specified geometry *geom*. The top-level geometry description includes a list of up to MaxColors (32) *color names*. A color *name* is a string whose interpretation is not specified by Xkb, neither is the *pixel* value's interpretation specified. All other geometry data structures refer to colors using their indices in this global list or pointers to colors in this list. *XkbAddGeomColor* returns NULL if any of the parameters is empty or if it was not able to allocate space for the color. To allocate space for an arbitrary number of colors to a geometry, call the *XkbAllocGeomColors* function.

To add one outline to an existing shape, call *XkbAddGeomOutline*.

```
XkbOutlinePtr XkbAddGeomOutline(shape, sz_points)
    XkbShapePtr   shape;          /* shape to be updated */
    int          sz_points;       /* number of points to be reserved */
```

An outline consists of an arbitrary number of points. *XkbAddGeomOutline* adds an outline to the specified *shape* by reserving *sz_points* points for it. The new outline is allocated and zeroed. *XkbAddGeomOutline* returns NULL if any of the parameters is empty or if it was not able to allocate space. To allocate space for an arbitrary number of outlines to a shape, call *XkbAllocGeomOutlines*.

To add a shape to a keyboard geometry, call *XkbAddGeomShape*.

```
XkbShapePtr XkbAddGeomShape(geom, name, sz_outlines)
    XkbGeometryPtr geom;          /* geometry to be updated */
    Atom         name;           /* name of the new shape */
    int          sz_outlines;    /* number of outlines to be reserved */
```

A geometry contains an arbitrary number of shapes, each of which is made up of an arbitrary number of outlines. *XkbAddGeomShape* adds a shape to a geometry *geom* by allocating space for *sz_outlines* outlines for it, and giving it the name specified by *name*. If a shape with name *name* already exists in the geometry, a pointer to the existing shape is returned. *XkbAddGeomShape* returns NULL if any of the parameters is empty or if it was not able to allocate space. To allocate space for an arbitrary number of geometry shapes, call *XkbAllocGeomShapes*.

To add one key at the end of an existing row of keys, call *XkbAddGeomKey*.

```
XkbKeyPtr XkbAddGeomKey(row)
    XkbRowPtr   row;           /* row to be updated */
```

Keys are grouped into rows. *XkbAddGeomKey* adds one key to the end of the specified *row*. The key is allocated and zeroed. *XkbAddGeomKey* returns NULL if *row* is empty or if it was not able to allocate space for the key. To allocate space for an arbitrary number of keys to a row, call *XkbAllocGeomKeys*.

To add one section to an existing keyboard geometry, call *XkbAddGeomSection*.

```
XkbSectionPtr XkbAddGeomSection(geom, name, sz_rows, sz_doodads, sz_overlays)
    XkbGeometryPtr geom;          /* geometry to be updated */
    Atom         name;           /* name of the new section */
    int          sz_rows;        /* number of rows to reserve in the section */
    int          sz_doodads;     /* number of doodads to reserve in the section */
    int          sz_overlays;    /* number of overlays to reserve in the section */
```

A keyboard geometry contains an arbitrary number of sections. *XkbAddGeomSection* adds one section to an existing keyboard geometry *geom*. The new section contains space for the number of rows, doodads, and overlays specified by *sz_rows*, *sz_doodads*, and *sz_overlays*. The new section is allocated and zeroed, and given the name specified by *name*. If a section with name *name* already exists in the geometry, a pointer to the existing section is returned. *XkbAddGeomSection* returns NULL if any of the parameters is empty or if it was not able to allocate space for the section. To allocate space for an arbitrary number of sections to a geometry, use *XkbAllocGeomSections*.

To add a row to a section, call *XkbAddGeomRow*.

```
XkbRowPtr XkbAddGeomRow(section, sz_keys)
    XkbSectionPtr section;       /* section to be updated */
    int          sz_keys;        /* number of keys to be reserved */
```

One of the components of a keyboard geometry section is one or more rows of keys. *XkbAddGeomRow* adds one row to the specified *section*. The newly created row contains space for the number of keys specified in *sz_keys*. They are allocated and zeroed, but otherwise uninitialized. *XkbAddGeomRow* returns NULL if any of the parameters is empty or if it was not able to allocate space for the row. To allocate space for an arbitrary number of rows to a section, call the *XkbAllocGeomRows* function.

To add one doodad to a section of a keyboard geometry, or to the top-level geometry, call *XkbAddGeomDoodad*.

```
XkbDoodadPtr XkbAddGeomDoodad(geom, section, name)
    XkbGeometryPtr geom;          /* geometry to which the doodad is added */
    XkbSectionPtr  section;       /* section, if any, to which the doodad is added */
    Atom           name;          /* name of the new doodad */
```

A *doodad* describes some visible aspect of the keyboard that is not a key and is not a section. *XkbAddGeomDoodad* adds a doodad with name specified by *name* to the geometry *geom* if *section* is NULL, or to the section of the geometry specified by *section* if *section* is not NULL. *XkbAddGeomDoodad* returns NULL if any of the parameters is empty or if it was not able to allocate space for the doodad. If there is already a doodad with the name *name* in the doodad array for the geometry (if *section* is NULL) or the section (if *section* is non-NULL), a pointer to that doodad is returned. To allocate space for an arbitrary number of doodads to a section, use the *XkbAllocGeomSectionDoodads* function. To allocate space for an arbitrary number of doodads to a keyboard geometry, use the *XkbAllocGeomDoodads* function.

To add one overlay to a section, call *XkbAddGeomOverlay*.

```
XkbOverlayPtr XkbAddGeomOverlay(section, name, sz_rows)
    XkbSectionPtr  section;       /* section to which an overlay will be added */
    Atom           name;          /* name of the overlay */
    int            sz_rows;        /* number of rows to reserve in the overlay */
```

XkbAddGeomOverlay adds an overlay with the specified name to the specified *section*. The new overlay is created with space allocated for *sz_rows* rows. If an overlay with name *name* already exists in the section, a pointer to the existing overlay is returned. *XkbAddGeomOverlay* returns NULL if any of the parameters is empty or if it was not able to allocate space for the overlay. To allocate space for an arbitrary number of overlays to a section, call the *XkbAllocGeomOverlay* function.

To add a row to an existing overlay, call *XkbAddGeomOverlayRow*.

```
XkbOverlayRowPtr XkbAddGeomOverlayRow(overlay, row_under, sz_keys)
    XkbOverlayPtr  overlay;        /* overlay to be updated */
    XkbRowPtr      row_under;      /* row to be overlaid in the section overlay overlays */
    int            sz_keys;        /* number of keys to reserve in the row */
```

XkbAddGeomOverlayRow adds one row to the *overlay*. The new row contains space for *sz_keys* keys. If *row_under* specifies a row that doesn't exist on the underlying section, *XkbAddGeomOverlayRow* returns NULL and doesn't change the overlay. *XkbAddGeomOverlayRow* returns NULL if any of the parameters is empty or if it was not able to allocate space for the overlay.

To add a key to an existing overlay row, call *XkbAddGeomOverlayKey*.

```
XkbOverlayKeyPtr XkbAddGeomOverlayKey(overlay, row, under)
    XkbOverlayPtr  overlay;        /* overlay to be updated */
    XkbRowPtr      row;           /* row in overlay to be updated */
    char *         under;         /* primary name of the key to be considered */
```

XkbAddGeomOverlayKey adds one key to the *row* in the *overlay*. If there is no key named *under* in the row of the underlying section, *XkbAddGeomOverlayKey* returns NULL.

13.11 Allocating and Freeing Geometry Components

Xkb provides a number of functions to allocate and free subcomponents of a keyboard geometry. Use these functions to create or modify keyboard geometries. Note that these functions merely allocate space for the new element(s), and it is up to you to fill in the values explicitly in your code. These allocation functions increase *sz_** but never touch *num_** (unless there is an allocation failure, in which case they reset both *sz_** and *num_** to zero). These functions return Success if they succeed, BadAlloc if they are not able to allocate space, or BadValue if a parameter is not as expected.

To allocate space for an arbitrary number of outlines to a shape, call *XkbAllocGeomOutlines*.

Status **XkbAllocGeomOutlines**(*shape*, *num_needed*)

XkbShapePtr	<i>shape</i> ; /* shape for which outlines should be allocated */
int	<i>num_needed</i> ; /* number of new outlines required */

XkbAllocGeomOutlines allocates space for *num_needed* outlines in the specified *shape*. The outlines are not initialized.

To free geometry outlines, call *XkbFreeGeomOutlines*.

void **XkbFreeGeomOutlines**(*shape*, *first*, *count*, *free_all*)

XkbShapePtr	<i>shape</i> ; /* shape in which outlines should be freed */
int	<i>first</i> ; /* first outline to be freed */
int	<i>count</i> ; /* number of outlines to be freed */
Bool	<i>free_all</i> /* True => all outlines are freed */

If *free_all* is True, all outlines are freed regardless of the value of *first* or *count*. Otherwise, *count* outlines are freed beginning with the one specified by *first*.

To allocate space for an arbitrary number of keys to a row, call *XkbAllocGeomKeys*.

Status **XkbAllocGeomKeys**(*row*, *num_needed*)

XkbRowPtr	<i>row</i> ; /* row to which keys should be allocated */
int	<i>num_needed</i> ; /* number of new keys required */

XkbAllocGeomKeys allocates *num_needed* keys and adds them to the *row*. No initialization of the keys is done.

To free geometry keys, call *XkbFreeGeomKeys*.

void **XkbFreeGeomKeys**(*row*, *first*, *count*, *free_all*)

XkbRowPtr	<i>row</i> ; /* row in which keys should be freed */
int	<i>first</i> ; /* first key to be freed */
int	<i>count</i> ; /* number of keys to be freed */
Bool	<i>free_all</i> /* True => all keys are freed */

If *free_all* is True, all keys are freed regardless of the value of *first* or *count*. Otherwise, *count* keys are freed beginning with the one specified by *first*.

To allocate geometry properties, call *XkbAllocGeomProps*.

Status **XkbAllocGeomProps**(*geom*, *num_needed*)

XkbGeometryPtr	<i>geom</i> ; /* geometry for which properties should be allocated */
int	<i>num_needed</i> ; /* number of new properties required */

XkbAllocGeomProps allocates space for *num_needed* properties and adds them to the specified geometry *geom*. No initialization of the properties is done. 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.

To free geometry properties, call *XkbFreeGeomProperties*.

```
void XkbFreeGeomProperties(geom, first, count, free_all)
    XkbGeometryPtr geom;          /* geometry in which properties should be freed */
    int           first;          /* first property to be freed */
    int           count;          /* number of properties to be freed */
    Bool          free_all        /* True => all properties are freed */
```

If *free_all* is True, all properties are freed regardless of the value of *first* or *count*. Otherwise, *count* properties are freed beginning with the one specified by *first*.

To allocate geometry key aliases, call *XkbAllocGeomKeyAliases*.

```
Status XkbAllocGeomKeyAliases(geom, num_needed)
    XkbGeometryPtr geom;          /* geometry for which key aliases should be allocated */
    int           num_needed; /* number of new key aliases required */
```

XkbAllocGeomKeyAliases allocates space for *num_needed* key aliases and adds them to the specified geometry *geom*. A key alias is a pair of strings that associates an alternate name for a key with the real name for that key.

To free geometry key aliases, call *XkbFreeGeomKeyAliases*.

```
void XkbFreeGeomKeyAliases(geom, first, count, free_all)
    XkbGeometryPtr geom;          /* geometry in which key aliases should be freed */
    int           first;          /* first key alias to be freed */
    int           count;          /* number of key aliases to be freed */
    Bool          free_all        /* True => all key aliases are freed */
```

If *free_all* is True, all aliases in the top level of the specified geometry *geom* are freed regardless of the value of *first* or *count*. Otherwise, *count* aliases in *geom* are freed beginning with the one specified by *first*.

To allocate geometry colors, call *XkbAllocGeomColors*.

```
Status XkbAllocGeomColors(geom, num_needed)
    XkbGeometryPtr geom;          /* geometry for which colors should be allocated */
    int           num_needed; /* number of new colors required */
```

XkbAllocGeomColors allocates space for *num_needed* colors and adds them to the specified geometry *geom*. A color name is a string whose interpretation is not specified by Xkb. All other geometry data structures refer to colors using their indices in this global list or pointers to colors in this list.

To free geometry colors, call *XkbFreeGeomColors*.

```
void XkbFreeGeomColors(geom, first, count, free_all)
    XkbGeometryPtr geom;          /* geometry in which colors should be freed */
    int           first;          /* first color to be freed */
    int           count;          /* number of colors to be freed */
    Bool          free_all;       /* True => all colors are freed */
```

If *free_all* is True, all colors are freed regardless of the value of *first* or *count*. Otherwise, *count* colors are freed beginning with the one specified by *first*.

To allocate points in an outline, call *XkbAllocGeomPoints*.

```
Status XkbAllocGeomPoints(outline, num_needed)
    XkbOutlinePtr outline;        /* outline for which points should be allocated */
    int           num_needed;    /* number of new points required */
```

XkbAllocGeomPoints allocates space for *num_needed* points in the specified *outline*. The points are not initialized.

To free points in a outline, call *XkbFreeGeomPoints*.

```
void XkbFreeGeomPoints(outline, first, count, free_all)
    XkbOutlinePtr outline;        /* outline in which points should be freed */
    int           first;          /* first point to be freed */
    int           count;          /* number of points to be freed */
    Bool          free_all;       /* True => all points are freed */
```

If *free_all* is True, all points are freed regardless of the value of *first* and *count*. Otherwise, the number of points specified by *count* are freed, beginning with the point specified by *first* in the specified outline.

To allocate space for an arbitrary number of geometry shapes, call *XkbAllocGeomShapes*.

```
Status XkbAllocGeomShapes(geom, num_needed)
    XkbGeometryPtr geom;          /* geometry for which shapes should be allocated */
    int           num_needed;    /* number of new shapes required */
```

XkbAllocGeomShapes allocates space for *num_needed* shapes in the specified geometry *geom*. The shapes are not initialized.

To free geometry shapes, call *XkbFreeGeomShapes*.

```
void XkbFreeGeomShapes(geom, first, count, free_all)
    XkbGeometryPtr geom;          /* geometry in which shapes should be freed */
    int           first;          /* first shape to be freed */
    int           count;          /* number of shapes to be freed */
    Bool          free_all;       /* True => all shapes are freed */
```

If *free_all* is True, all shapes in the geometry are freed regardless of the values of *first* and *count*. Otherwise, *count* shapes are freed, beginning with the shape specified by *first*.

To allocate geometry sections, call *XkbAllocGeomSections*.

```
Status XkbAllocGeomSections(geom, num_needed)
    XkbGeometryPtr geom;          /* geometry for which sections should be allocated */
    int           num_needed;    /* number of new sections required */
```

XkbAllocGeomSections allocates *num_needed* sections and adds them to the geometry *geom*. No initialization of the sections is done.

To free geometry sections, call *XkbFreeGeomSections*.

```
void XkbFreeGeomSections(geom, first, count, free_all)
    XkbGeometryPtr geom;          /* geometry in which sections should be freed */
    int            first;          /* first section to be freed. */
    int            count;          /* number of sections to be freed */
    Bool           free_all        /* True => all sections are freed */
```

If *free_all* is True, all sections are freed regardless of the value of *first* and *count*. Otherwise, the number of sections specified by *count* are freed, beginning with the section specified by *first* in the specified geometry.

To allocate rows in a section, call *XkbAllocGeomRows*.

```
Status XkbAllocGeomRows(section, num_needed)
    XkbSectionPtr section;        /* section for which rows should be allocated */
    int           num_needed;     /* number of new rows required */
```

XkbAllocGeomRows allocates *num_needed* rows and adds them to the *section*. No initialization of the rows is done.

To free rows in a section, call *XkbFreeGeomRows*.

```
void XkbFreeGeomRows(section, first, count, free_all)
    XkbSectionPtr section;        /* section in which rows should be freed */
    int           first;          /* first row to be freed. */
    int           count;          /* number of rows to be freed */
    Bool           free_all        /* True => all rows are freed */
```

If *free_all* is True, all rows are freed regardless of the value of *first* and *count*. Otherwise, the number of rows specified by *count* are freed, beginning with the row specified by *first* in the specified section.

To allocate overlays in a section, call *XkbAllocGeomOverlays*.

```
Status XkbAllocGeomOverlays(section, num_needed)
    XkbSectionPtr section;        /* section for which overlays should be allocated */
    int           num_needed;     /* number of new overlays required */
```

XkbAllocGeomRows allocates *num_needed* overlays and adds them to the *section*. No initialization of the overlays is done.

To free rows in an section, call *XkbFreeGeomOverlays*.

```
void XkbFreeGeomOverlays(section, first, count, free_all)
    XkbSectionPtr section;        /* section in which overlays should be freed */
    int           first;          /* first overlay to be freed. */
    int           count;          /* number of overlays to be freed */
    Bool           free_all        /* True => all overlays are freed */
```

If *free_all* is True, all overlays are freed regardless of the value of *first* and *count*. Otherwise, the number of overlays specified by *count* are freed, beginning with the overlay specified by *first* in the specified section.

To allocate rows in a overlay, call *XkbAllocGeomOverlayRows*.

Status **XkbAllocGeomOverlayRows**(*overlay*, *num_needed*)
 XkbSectionPtr *overlay*; /* section for which rows should be allocated */
 int *num_needed*; /* number of new rows required */

XkbAllocGeomOverlayRows allocates *num_needed* rows and adds them to the *overlay*. No initialization of the rows is done.

To free rows in an overlay, call *XkbFreeGeomOverlayRows*.

void **XkbFreeGeomOverlayRows**(*overlay*, *first*, *count*, *free_all*)
 XkbSectionPtr *overlay*; /* section in which rows should be freed */
 int *first*; /* first row to be freed. */
 int *count*; /* number of rows to be freed */
 Bool *free_all* /* True => all rows are freed */

If *free_all* is True, all rows are freed regardless of the value of *first* and *count*. Otherwise, the number of rows specified by *count* are freed, beginning with the row specified by *first* in the specified overlay.

To allocate keys in an overlay row, call *XkbAllocGeomOverlayKeys*.

Status **XkbAllocGeomOverlayKeys**(*row*, *num_needed*)
 XkbRowPtr *row*; /* section for which rows should be allocated */
 int *num_needed*; /* number of new rows required */

XkbAllocGeomOverlayKeys allocates *num_needed* keys and adds them to the *row*. No initialization of the keys is done.

To free keys in an overlay row, call *XkbFreeGeomOverlayKeys*.

void **XkbFreeGeomOverlayKeys**(*row*, *first*, *count*, *free_all*)
 XkbRowPtr *row*; /* row in which keys should be freed */
 int *first*; /* first key to be freed. */
 int *count*; /* number of keys to be freed */
 Bool *free_all* /* True => all keys are freed */

If *free_all* is True, all keys are freed regardless of the value of *first* and *count*. Otherwise, the number of keys specified by *count* are freed, beginning with the key specified by *first* in the specified row.

To allocate doodads that are global to a keyboard geometry, call *XkbAllocGeomDoodads*.

Status **XkbAllocGeomDoodads**(*geom*, *num_needed*)
 XkbGeometryPtr *geom*; /* geometry for which doodads should be allocated */
 int *num_needed*; /* number of new doodads required */

XkbAllocGeomDoodads allocates *num_needed* doodads and adds them to the specified geometry *geom*. No initialization of the doodads is done.

To allocate doodads that are specific to a section, call *XkbAllocGeomSectionDoodads*.

Status **XkbAllocGeomSectionDoodads**(*section*, *num_needed*)
 XkbSectionPtr *section*; /* section for which doodads should be allocated */
 int *num_needed*; /* number of new doodads required */

XkbAllocGeomSectionDoodads allocates *num_needed* doodads and adds them to the specified *section*. No initialization of the doodads is done.

To free geometry doodads, call *XkbFreeGeomDoodads*.

```
void XkbFreeGeomDoodads(doodads, count, free_all)
    XkbDoodadPtr   doodads;      /* doodads to be freed */
    int            count;        /* number of doodads to be freed */
    Bool           free_all;     /* True => all doodads are freed */
```

If *free_all* is True, all doodads in the array are freed, regardless of the value of *count*. Otherwise, *count* doodads are freed.

To allocate an entire geometry, call *XkbAllocGeometry*.

```
Status XkbAllocGeometry(xkb, sizes)
    XkbDescPtr      xkb; /* keyboard description for which geometry is to be allocated */
    XkbGeometrySizesPtr sizes; /* initial sizes for all geometry components */
```

XkbAllocGeometry allocates a keyboard geometry and adds it to the keyboard description specified by *xkb*. The keyboard description should be obtained via the *XkbGetKeyboard* or *XkbAllocKeyboard* functions. The *sizes* parameter specifies the number of elements to be reserved for the subcomponents of the keyboard geometry and can be zero or more. These subcomponents include the *properties*, *colors*, *shapes*, *sections*, and *doodads*.

To free an entire geometry, call *XkbFreeGeometry*.

```
void XkbFreeGeometry(geom, which, free_all)
    XkbGeometryPtr geom;      /* geometry to be freed */
    unsigned int   which;     /* mask of geometry components to be freed */
    Bool          free_all;   /* True => the entire geometry is freed. */
```

The values of *which* and *free_all* determine how much of the specified geometry is freed. The valid values for *which* are:

```
#define XkbGeomPropertiesMask  (1<<0)
#define XkbGeomColorsMask       (1<<1)
#define XkbGeomShapesMask        (1<<2)
#define XkbGeomSectionsMask      (1<<3)
#define XkbGeomDoodadsMask       (1<<4)
#define XkbGeomAllMask           (0x1f)
```

If *free_all* is True, the entire geometry is freed regardless of the value of *which*. Otherwise, the portions of the geometry specified by *which* are freed.

14 Xkb Keyboard Mapping

The Xkb keyboard mapping contains all the information the server and clients need to interpret key events. This chapter provides an overview of the terminology used to describe an Xkb keyboard mapping, and introduces common utilities for manipulating the keyboard mapping.

The mapping consists of two components, a server map and a client map. The *client* map is the collection of information a client needs to interpret key events from the keyboard. It contains a global list of key types and an array of key symbol maps, each of which describes the symbols bound to a key and the rules to be used to interpret those symbols. The *server* map contains the information the server needs to interpret key events. This includes actions and behaviors for each key, explicit components for a key and the virtual modifiers and the per-key virtual modifier mapping.

For detailed information on particular components of the keyboard map, refer to Chapter 15, “Xkb Client Keyboard Mapping” and Chapter 16, “Xkb Server Keyboard Mapping.”

14.1 Notation and Terminology

The graphic characters or control functions that may be accessed by one key are logically arranged in groups and levels, where *group* and *level* are defined as in the ISO9995 standard:

- | | |
|--------|--|
| Group: | A logical state of a keyboard providing access to a collection of graphic characters. Usually these graphic characters logically belong together and may be arranged on several levels within a group. |
| Level: | One of several states (normally 2 or 3) governing which graphic character is produced when a graphic key is actuated. In certain cases the level may also affect function keys. |

The above definitions, taken from the ISO standard, refer to graphic keys and characters. In the context of Xkb, Group and Level are not constrained to graphic keys and characters; they may be used with any key to access any character the key is capable of generating.

Level is often referred to as “Shift Level”. Levels are numbered sequentially starting at one.

Note 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 (that is, the NumLock virtual modifier) also selects shift level 2.

For example, consider the following key (the gray characters indicate symbols that are implied or expected but are not actually engraved on the key):

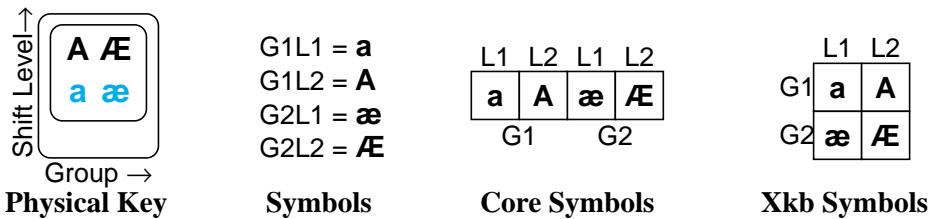


Figure 14.1 Shift Levels and Groups

This key has two groups, indicated by the columns, and each group has two shift levels. For the first group (Group1), the symbol shift level one is **a**, and the symbol for shift level two is **A**. For the second group, the symbol for shift level one is **æ**, and the symbol for shift level two is **Æ**.

14.1.1 Core Implementation

The standard interpretation rules for the core X keymap only allow clients to access keys such as the one shown in Figure 14.1. That is, clients using the standard interpretation rules can only access one of four keysyms for any given `KeyPress` event — two different symbols in two different groups.

In general, the `Shift` modifier, the `Lock` modifier, and the modifier bound to the `Num_Lock` key are used to change between shift level 1 and shift level 2. To switch between groups, the core implementation uses the modifier bound to the `Mode_switch` key. When the `Mode_switch` modifier is set, the keyboard is logically in Group 2. When the `Mode_switch` modifier is not set, the keyboard is logically in Group 1.

The core implementation does not clearly specify the behavior of keys. For example, the locking behavior of the `CapsLock` and `Num_Lock` keys depends upon the vendor.

14.1.2 Xkb Implementation

Xkb extends the core implementation by providing access to up to four keyboard groups with up to 63 shift levels per key¹. In addition, Xkb provides precise specifications regarding the behavior of keys. In Xkb, modifier state and the current group are independent (with the exception of compatibility mapping, discussed in Chapter 17.)

Xkb handles switching between groups via key actions, independent of any modifier state information. Key actions are in the server map component and are described in detail in section Note on page 145.

Xkb handles shift levels by associating a key type with each group on each key. Each key type defines the shift levels available for the groups on keys of its type, and specifies the modifier combinations necessary to access each level.

1. The core implementation restricts the number of symbols per key to 255. With four groups, this allows for up to 63 symbols (or shift levels) per group. Most keys will only have a few shift levels.

For example, Xkb allows key types where the Control modifier can be used to access the shift level two of a key. Key types are in the client map component and are described in detail in section 15.2 on page 127.

Xkb provides precise specification of the behavior of a key using key behaviors. Key behaviors are in the server map component and are described in detail in section 16.2 on page 161.

14.2 Getting Map Components from the Server

Xkb provides two routines to obtain the keyboard mapping components from the server. The first routine, *XkbGetMap*, allocates an *XkbDescRec* structure, retrieves mapping components from the server, and stores them in the *XkbDescRec* structure it just allocated. The second routine, *XkbGetUpdatedMap*, retrieves mapping components from the server and stores them in a *XkbDescRec* structure that has previously been allocated.

To allocate an *XkbDescRec* structure and populate it with the server's keyboard client map and server map, call *XkbGetMap*. *XkbGetMap* is similar to *XkbGetKeyboard* (see section 6.2), but is used only for obtaining the address of an *XkbDescRec* structure that is populated with keyboard mapping components. It allows finer control over which sub-structures of the keyboard mapping components are to be populated. *XkbGetKeyboard* always returns fully-populated components, while *XkbGetMap* can be instructed to return a partially-populated component.

```
XkbDescPtr XkbGetMap(display, which, device_spec)
    Display *          display;      /* connection to X server */
    unsigned int        which;       /* mask selecting sub-components to populate */
    unsigned int        device_spec; /* device_id, or XkbUseCoreKbd */
```

The *which* mask is a bitwise inclusive OR of the masks defined in Table 14.1 on page 119. Only those portions of the keyboard server map and the keyboard client maps that are specified in *which* are allocated and populated.

In addition to allocating and obtaining the server map and the client map, *XkbGetMap* also sets the *device_spec*, the *min_key_code*, and *max_key_code* fields of the keyboard description.

XkbGetMap is synchronous; it queries the server for the desired information, waits for a reply, and then returns. If successful, *XkbGetMap* returns a pointer to the *XkbDescRec* structure it allocated. If unsuccessful, *XkbGetMap* returns NULL. When unsuccessful, one of the following protocol errors is also generated: *BadAlloc* (unable to allocate the *XkbDescRec* structure), *BadValue* (some mask bits in *which* are undefined), or *BadImplementation* (a compatible version of the Xkb extension is not available in the server). To free the returned data, use *XkbFreeClientMap*.

Xkb also provides convenience routines to get partial component definitions from the server. These functions are specified in the “convenience functions” column in Table 14.1. Refer to the sections listed in the table for more information on these routines.

Table 14.1 Xkb Mapping Component Masks and Convenience Functions

Mask	Value Map	Fields	Convenience Functions	Section
XkbKeyTypesMask	(1<<0) client	types size_types num_types	XkbGetKeyTypes XkbResizeKeyType XkbCopyKeyType XkbCopyKeyTypes	15.2
XkbKeySymsMask	(1<<1) client	syms size_syms num_syms key_sym_map	XkbGetKeySyms XkbResizeKeySyms XkbChangeTypesOfKey	15.3
XkbModifierMapMask	(1<<2) client	modmap	XkbGetKeyModifierMap	15.4
XkbExplicitComponentsMask	(1<<3) server	explicit	XkbGetKeyExplicitComponents	16.3
XkbKeyActionsMask	(1<<4) server	key_acts acts num_acts size_acts	XkbGetKeyActions XkbResizeKeyActions	16.1
XkbKeyBehaviorsMask	(1<<5) server	behaviors	XkbGetKeyBehaviors	16.2
XkbVirtualModsMask	(1<<6) server	vmods	XkbGetVirtualMods	16.4
XkbVirtualModMapMask	(1<<7) server	vmodmap	XkbGetVirtualModMap	16.4

Xkb defines combinations of these masks for convenience:

```
#define XkbResizableInfoMask          (XkbKeyTypesMask)
#define XkbAllClientInfoMask          (XkbKeyTypesMask | XkbKeySymsMask |
XkbModifierMapMask)
#define XkbAllServerInfoMask          (XkbExplicitComponentsMask |
XkbKeyActionsMask|XkbKeyBehaviorsMask |
XkbVirtualModsMask | XkbVirtualModMapMask)
#define XkbAllMapComponentsMask        (XkbAllClientInfoMask|XkbAllServerInfoMask)
```

Key types, symbol maps, and actions are all interrelated: changing one requires changes in the others. The convenience functions make it easier to edit these components and handle the interdependencies.

To update the client or server map information in an existing keyboard description, use *XkbGetUpdatedMap*.

Status **XkbGetUpdatedMap**(*display*, *which*, *xkb*)

Display *	<i>display</i> ;	/* connection to X server */
unsigned int	<i>which</i> ;	/* mask selecting sub-components to populate */
XkbDescPtr	<i>xkb</i> ;	/* keyboard description to be updated */

The *which* parameter is a bitwise inclusive OR of the masks in Table 14.1. If the needed components of the *xkb* structure are not already allocated, *XkbGetUpdatedMap* allocates them. *XkbGetUpdatedMap* fetches the requested information for the device specified in the *XkbDescRec* passed in the *xkb* parameter.

XkbGetUpdatedMap is synchronous; it queries the server for the desired information, waits for a reply, and then returns. If successful, *XkbGetUpdatedMap* returns Success. If unsuccessful, *XkbGetUpdatedMap* returns one of the following: BadAlloc (unable to allocate a component in the *XkbDescRec* structure), BadValue (some mask bits in *which* are undefined), BadImplementation (a compatible version of the Xkb extension is not available in the server or the reply from the server was invalid).

14.3 Changing Map Components in the Server

There are two ways to make changes to map components: either change a local copy of the keyboard map and call *XkbSetMap* to send the modified map to the server, or, to reduce network traffic, use an *XkbMapChangesRec* structure and call *XkbChangeMap*.

```
Bool XkbSetMap(dpy, which, xkb)
    Display *          dpy;      /* connection to X server */
    unsigned int        which;    /* mask selecting sub-components to update */
    XkbDescPtr         xkb;     /* keyboard description from which new values taken */
```

Use *XkbSetMap* to send a complete new set of values for entire components (for example, all symbols, all actions, etc.) to the server. The *which* parameter specifies the components to be sent to the server, and is a bitwise inclusive OR of the masks listed in Table 14.1. The *xkb* parameter is a pointer to an *XkbDescRec* structure, and contains the information to be copied to the server. For each bit set in the *which* parameter, *XkbSetMap* takes the corresponding structure values from the *xkb* parameter and sends it to the server specified by *dpy*.

If any components specified by *which* are not present in the *xkb* parameter, *XkbSetMap* returns False. Otherwise, it sends the update request to the server and returns True. *XkbSetMap* can generate BadAlloc, BadLength, and BadValue protocol errors.

Key types, symbol maps, and actions are all interrelated; changing one requires changes in the others. Xkb provides functions to make it easier to edit these components and handle the interdependencies. Table 14.1 lists these helper functions and provides a pointer to where they are defined

14.3.1 The *XkbMapChangesRec* Structure

Use the *XkbMapChangesRec* structure to identify and track partial modifications to the mapping components, and to reduce the amount of traffic between the server and clients.

```
typedef struct _XkbMapChanges {
    unsigned short    changed;           /* identifies valid components in structure */
    KeyCode          min_key_code;       /* lowest numbered keycode for device */
    KeyCode          max_key_code;       /* highest numbered keycode for device */
    unsigned char    first_type;        /* index of first key type modified */
    unsigned char    num_types;         /* # types modified */
    KeyCode          first_key_sym;     /* first key whose key_sym_map changed */
    unsigned char    num_key_syms;      /* # key_sym_map entries changed */
    KeyCode          first_key_act;     /* first key whose keyActs entry changed */
    unsigned char    num_key Acts;      /* # key_acts entries changed */
    KeyCode          first_key_behavior; /* first key whose behaviors changed */
    unsigned char    num_key_behaviors; /* # behaviors entries changed */
    KeyCode          first_key_explicit; /* first key whose explicit entry changed */
```

```

    unsigned char      num_key_explicit;    /* # explicit entries changed */
    KeyCode           first_modmap_key;   /* first key whose modmap entry changed */
    unsigned char      num_modmap_keys;    /* # modmap entries changed */
    KeyCode           first_vmodmap_key;  /* first key whose vmodmap changed */
    unsigned char      num_vmodmap_keys;   /* # vmodmap entries changed */
    unsigned char      pad1;              /* reserved */
    unsigned short     vmods;             /* mask indicating which vmods changed */
} XkbMapChangesRec,*XkbMapChangesPtr;

```

The *changed* field identifies the map components that have changed in an XkbDescRec structure, and may contain any of the bits in Table 14.1 on page 119, which are also shown in Table 14.2. Every 1 bit in *changed* also identifies which other fields in the XkbMapChangesRec structure contain valid values, as indicated in Table 14.2. The *min_key_code* and *max_key_code* fields are for reference only; they are ignored on any requests sent to the server, and are always updated by the server whenever it returns the data for an XkbMapChangesRec.

Table 14.2 XkbMapChangesRec Masks

<i>Mask</i>	<i>Valid XkbMapChangesRec Fields</i>	<i>XkbDescRec Field Containing Changed Data</i>
XkbKeyTypesMask	first_type, num_types	map->type[first_type] .. map->type[first_type + num_types - 1]
XkbKeySymsMask	first_key_sym, num_key_syms	map->key_sym_map[first_key_sym] .. map->key_sym_map[first_key_sym + num_key_syms - 1]
XkbModifierMapMask	first_modmap_key, num_modmap_keys	map->modmap[first_modmap_key] .. map->modmap[first_modmap_key + num_modmap_keys - 1]
XkbExplicitComponentsMask	first_key_explicit, num_key_explicit	server->explicit[first_key_explicit] .. server->explicit[first_key_explicit + num_key_explicit - 1]
XkbKeyActionsMask	first_key_act, num_key_acts	server->key_acts[first_key_act] .. server->key_acts[first_key_act + num_key_acts - 1]
XkbKeyBehaviorsMask	first_key_behavior, num_key_behaviors	server->behaviors[first_key_behavior] .. server->behaviors[first_key_behavior + num_key_behaviors - 1]
XkbVirtualModsMask	vmods	server->vmods[*]
XkbVirtualModMapMask	first_vmodmap_key, num_vmodmap_keys	server->vmodmap[first_vmodmap_key] .. server->vmodmap[first_vmodmap_key + num_vmodmap_keys - 1]

To update only partial components of a keyboard description, modify the appropriate fields in the server and map components of a local copy of the keyboard description, then

call *XkbChangeMap* with an *XkbMapChangesRec* structure indicating which components have changed.

```
Bool XkbChangeMap(dpy, xkb, changes)
    Display *          dpy;           /* connection to X server */
    XkbDescPtr        xkb;           /* keyboard description from which new values taken */
    XkbMapChangesPtr  changes;        /*identifies component parts to update */
```

XkbChangeMap copies any components specified by the *changes* structure from the keyboard description, *xkb*, to the X server specified by *dp*y.

If any components specified by *changes* are not present in the *xkb* parameter, *XkbChangeMap* returns False. Otherwise, it sends a request to the server and returns True.

XkbChangeMap can generate BadAlloc, BadLength, and BadValue protocol errors.

14.4 Tracking Changes to Map Components

The Xkb extension reports *XkbMapNotify* events to clients wanting notification whenever a map component of the Xkb description for a device changes. There are many different types of Xkb keyboard map changes. Xkb uses an event detail mask to identify each type of change. The event detail masks are identical to the masks listed in Table 14.1 on page 119.

To receive *XkbMapNotify* events under all possible conditions, call *XkbSelectEvents* (see section 4.3) and pass *XkbMapNotifyMask* in both *bits_to_change* and *values_for_bits*.

To receive *XkbMapNotify* events only under certain conditions, call *XkbSelectEventDetails* using *XkbMapNotify* as the *event_type* and specifying the desired map changes in *bits_to_change* and *values_for_bits* using mask bits from Table 14.1.

The structure for *XkbMapNotify* events is:

```
typedef struct {
    int             type;                  /* Xkb extension base event code */
    unsigned long   serial;                /* X server serial number for event */
    Bool            send_event;            /* True => synthetically generated */
    Display *       display;               /* server connection where event generated */
    Time            time;                 /* server time when event generated */
    int             xkb_type;              /* XkbMapNotify */
    int             device;                /* Xkb device id, will not be XkbUseCoreKbd */
    unsigned int    changed;               /* identifies valid fields in rest of event */
    unsigned int    resized;               /* reserved */
    int             first_type;            /* index of first key type modified */
    int             num_types;              /* # types modified */
    KeyCode         min_key_code;          /* minimum keycode for device */
    KeyCode         max_key_code;          /* maximum keycode for device */
    KeyCode         first_key_sym;          /* first key whose key_sym_map changed */
    KeyCode         first_key_act;          /* first key whose keyActs entry changed */
    KeyCode         first_key_behavior;      /* first key whose behaviors changed */
    KeyCode         first_key_explicit;     /* first key whose explicit entry changed */
    KeyCode         first_modmap_key;       /* first key whose modmap entry changed */
```

```

KeyCode      first_vmodmap_key; /* # modmap entries changed */
int          num_key_syms;    /* # key_sym_map entries changed */
int          num_key_acts;    /* # key_acts entries changed */
int          num_key_behaviors; /* # behaviors entries changed */
int          num_key_explicit; /* # explicit entries changed */
int          num_modmap_keys; /* # modmap entries changed */
int          num_vmodmap_keys; /* # vmodmap entries changed */
unsigned int  vmods;          /* mask indicating which vmods changed */
} XkbMapNotifyEvent;

```

The *changed* field specifies the map components that have changed and is the bitwise inclusive OR of the mask bits defined in Table 14.1 on page 119. The other fields in this event are interpreted as the like-named fields in an *XkbMapChangesRec* (see section 14.3.1). The *XkbMapNotifyEvent* structure also has an additional *resized* field that is reserved for future use.

14.5 Allocating and Freeing Client and Server Maps

Calling *XkbGetMap* (see section 14.2) should be sufficient for most applications to get client and server maps. As a result, most applications do not need to directly allocate client and server maps.

If you change the number of key types or construct map components without loading the necessary components from the X server, do not allocate any map components directly using *malloc* or *Xmalloc*. Instead, use the Xkb allocators, *XkbAllocClientMap* and *XkbAllocServerMap*.

Similarly, use the Xkb destructors, *XkbFreeClientMap* and *XkbFreeServerMap* instead of *free* or *Xfree*.

14.5.1 Allocating an Empty Client Map

To allocate and initialize an empty client map description record, call *XkbAllocClientMap*.

Status **XkbAllocClientMap**(*xkb*, *which*, *type_count*)

<i>XkbDescPtr</i>	<i>xkb</i> ; /* keyboard description in which to allocate client map */
<i>unsigned int</i>	<i>which</i> ; /* mask selecting map components to allocate */
<i>unsigned int</i>	<i>type_count</i> ; /* value of <i>num_types</i> field in map to be allocated */

XkbAllocClientMap allocates and initializes an empty client map in the *map* field of the keyboard description specified by *xkb*. The *which* parameter specifies the particular components of the client map structure to allocate, and is a mask composed by a bitwise inclusive OR of one or more of the masks shown in Table 14.3.

Table 14.3 XkbAllocClientMap Masks

<i>Mask</i>	<i>Effect</i>
<i>XkbKeyTypesMask</i>	The <i>type_count</i> field specifies the number of entries to pre-allocate for the <i>types</i> field of the client map. If the <i>type_count</i> field is less than <i>XkbNumRequiredTypes</i> (see section 15.2.1), returns <i>BadValue</i> .

Table 14.3 XkbAllocClientMap Masks

<i>Mask</i>	<i>Effect</i>
XkbKeySymsMask	The <i>min_key_code</i> and <i>max_key_code</i> fields of the <i>xkb</i> parameter are used to allocate the <i>syms</i> and <i>key_sym_map</i> fields of the client map. The fields are allocated to contain the maximum number of entries necessary for <i>max_key_code - min_key_code + 1</i> keys.
XkbModifierMapMask	The <i>min_key_code</i> and <i>max_key_code</i> fields of the <i>xkb</i> parameter are used to allocate the <i>modmap</i> field of the client map. The field is allocated to contain the maximum number of entries necessary for <i>max_key_code - min_key_code + 1</i> keys.

Note The *min_key_code* and *max_key_code* fields of the *xkb* parameter must be legal values if the XkbKeySymsMask or XkbModifierMapMask masks are set in the *which* parameter. If they are not valid, *XkbAllocClientMap* returns *BadValue*.

If the client map of the keyboard description is not *NULL*, and any fields are already allocated in the client map, *XkbAllocClientMap* does not overwrite the existing values; it simply ignores that part of the request. The only exception is the *types* array. If *type_count* is greater than the current *num_types* field of the client map, *XkbAllocClientMap* resizes the *types* array and resets the *num_types* field accordingly.

If *XkbAllocClientMap* is successful, it returns *Success*. Otherwise, it can return either *BadMatch*, *BadAlloc*, or *BadValue* errors.

14.5.2 Freeing a Client Map

To free memory used by the client map member of an *XkbDescRec* structure, use *XkbFreeClientMap*.

```
void XkbFreeClientMap(xkb, which, free_all)
    XkbDescPtr xkb;          /* keyboard description containing client map to free */
    unsigned int which;       /* mask identifying components of map to free */
    Bool        free_all;     /* True => free all client components and map itself */
```

XkbFreeClientMap frees the components of client map specified by *which* in the *XkbDescRec* structure specified by the *xkb* parameter, and sets the corresponding structure component values to *NULL*. The *which* parameter specifies a combination of the client map masks shown in Table 14.3.

If *free_all* is *True*, *which* is ignored; *XkbFreeClientMap* frees every non-*NULL* structure component in the client map, frees the *XkbClientMapRec* structure referenced by the *map* member of the *xkb* parameter, and sets the *map* member to *NULL*.

14.5.3 Allocating an Empty Server Map

To allocate and initialize an empty server map description record, use *XkbAllocServerMap*.

```
Status XkbAllocServerMap(xkb, which, count_acts)
    XkbDescPtr   xkb;          /* keyboard description in which to allocate server map */
    unsigned int  which;       /* mask selecting map components to allocate */
    unsigned int  count_acts;  /* value of num_acts field in map to be allocated */
```

XkbAllocServerMap allocates and initializes an empty server map in the *server* field of the keyboard description specified by *xkb*. The *which* parameter specifies the particular components of the server map structure to allocate, as specified in Table 14.4.

Table 14.4 XkbAllocServerMap Masks

<i>Mask</i>	<i>Effect</i>
XkbExplicitComponentsMask	The <i>min_key_code</i> and <i>max_key_code</i> fields of the <i>xkb</i> parameter are used to allocate the <i>explicit</i> field of the server map.
XkbKeyActionsMask	The <i>min_key_code</i> and <i>max_key_code</i> fields of the <i>xkb</i> parameter are used to allocate the <i>key_acts</i> field of the server map. The <i>count_acts</i> parameter is used to allocate the <i>acts</i> field of the server map.
XkbKeyBehaviorsMask	The <i>min_key_code</i> and <i>max_key_code</i> fields of the <i>xkb</i> parameter are used to allocate the <i>behaviors</i> field of the server map.
XkbVirtualModMapMask	The <i>min_key_code</i> and <i>max_key_code</i> fields of the <i>xkb</i> parameter are used to allocate the <i>vmodmap</i> field of the server map.

Note The *min_key_code* and *max_key_code* fields of the *xkb* parameter must be legal values. If they are not valid, *XkbAllocServerMap* returns *BadValue*.

If the server map of the keyboard description is not *NULL* and any fields are already allocated in the server map, *XkbAllocServerMap* does not overwrite the existing values. The only exception is with the *acts* array. If the *count_acts* parameter is greater than the current *num_acts* field of the server map, *XkbAllocServerMap* resizes the *acts* array and resets the *num_acts* field accordingly.

If *XkbAllocServerMap* is successful, it returns *Success*. Otherwise, it can return either *BadMatch* or *BadAlloc* errors.

14.5.4 Freeing a Server Map

To free memory used by the server member of an *XkbDescRec* structure, use *XkbFreeServerMap*.

```
void XkbFreeServerMap(xkb, which, free_all)
    XkbDescPtr xkb;           /* keyboard description containing server map to free */
    unsigned int which;        /* mask identifying components of map to free */
    Bool free_all;           /* True => free all server map components and server itself */
```

The *XkbFreeServerMap* function frees the specified components of server map in the *XkbDescRec* structure specified by the *xkb* parameter, and sets the corresponding structure component values to *NULL*. The *which* parameter specifies a combination of the server map masks and is a bitwise inclusive OR of the masks listed in Table 14.4. If *free_all* is *True*, *which* is ignored and *XkbFreeServerMap* frees every non-*NULL* structure component in the server map, frees the *XkbServerMapRec* structure referenced by the *server* member of the *xkb* parameter, and sets the *server* member to *NULL*.

15 Xkb Client Keyboard Mapping

The Xkb client map for a keyboard is the collection of information a client needs to interpret key events from the keyboard. It contains a global list of key types and an array of key symbol maps, each of which describes the symbols bound to a key and the rules to be used to interpret those symbols.

Figure 15.1 shows the relationships between elements in the client map:

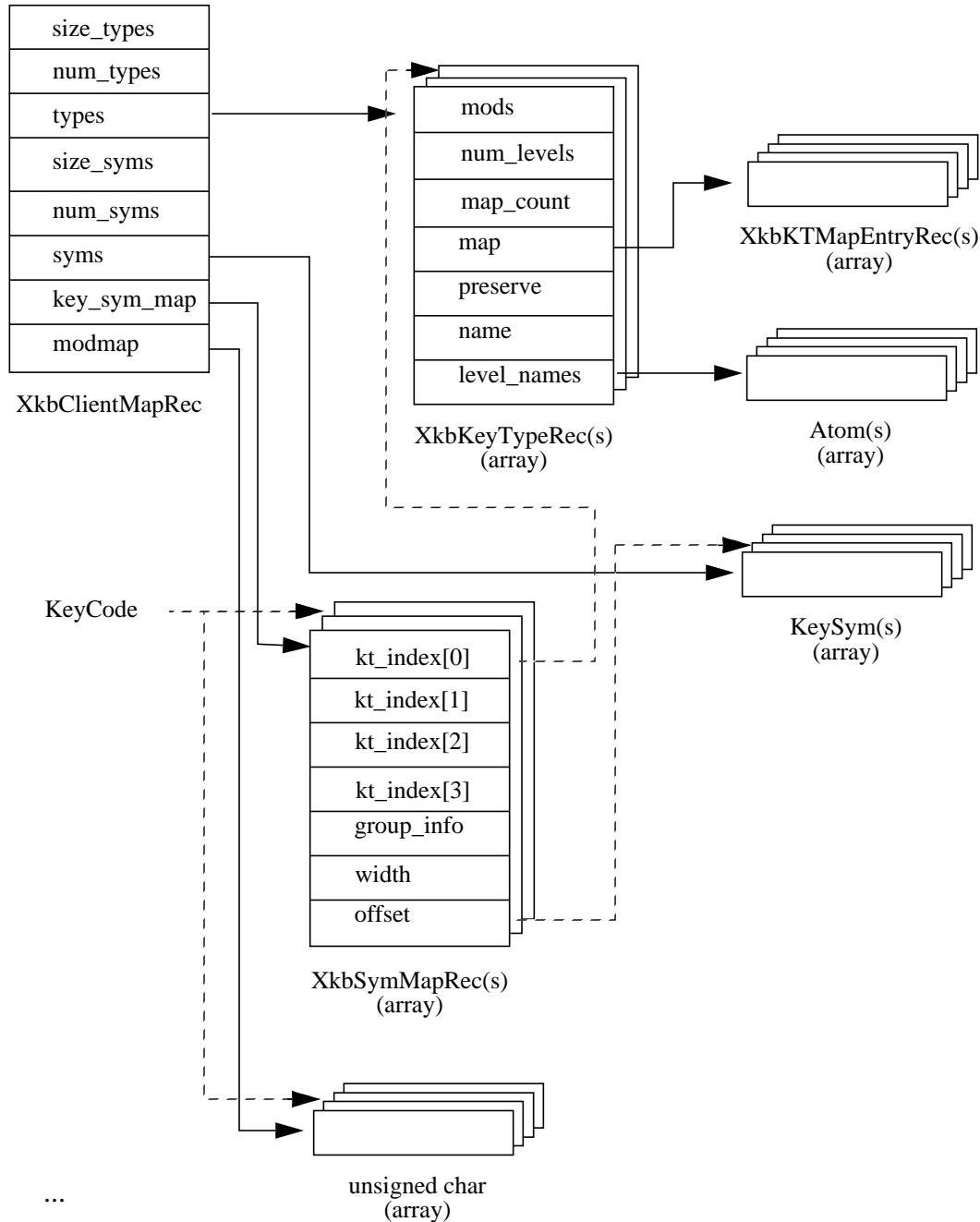


Figure 15.1 Xkb Client Map

15.1 The XkbClientMapRec Structure

The *map* field of the complete Xkb keyboard description (see section 6.1) is a pointer to the Xkb client map, which is of type XkbClientMapRec:

```
typedef struct {
    unsigned char      size_types;          /* Client Map */
    unsigned char      num_types;           /* # occupied entries in types */
    XkbKeyTypePtr     types;               /* # entries in types */
    unsigned short     size_syms;           /* vector of key types used by this keymap */
    unsigned short     num_syms;            /* length of the syms array */
    KeySym *           syms;                /* # entries in syms */
    XkbSymMapPtr       key_sym_map;         /* linear 2d tables of keysyms, 1 per key */
    unsigned char *    modmap;              /* 1 per keycode, maps keycode to syms */
} XkbClientMapRec, *XkbClientMapPtr;
```

The following sections describe each of the elements of the XkbClientMapRec structure in more detail.

15.2 Key Types

Key types are used to determine the shift level of a key given the current state of the keyboard. The set of all possible key types for the Xkb keyboard description are held in the *types* field of the client map, whose total size is stored in *size_types*, and whose total number of valid entries is stored in *num_types*. Key types are defined using the following structures:

```
typedef struct {                                     /* Key Type */
    XkbModsRec      mods;               /* modifiers used to compute shift level */
    unsigned char    num_levels;         /* total # shift levels, do not modify directly */
    unsigned char    map_count;          /* # entries in map, preserve (if non-NULL) */
    XkbKTMapEntryPtr map;               /* vector of modifiers for each shift level */
    XkbModsPtr       preserve;           /* mods to preserve for corresponding map entry */
    Atom             name;               /* name of key type */
    Atom *           level_names;        /* array of names of each shift level */
} XkbKeyTypeRec, *XkbKeyTypePtr;

typedef struct {                                     /* Modifiers for a key type */
    Bool             active;              /* True => entry active when determining shift level */
    unsigned char    level;               /* shift level if modifiers match mods */
    XkbModsRec       mods;               /* mods needed for this level to be selected */
} XkbKTMapEntryRec, *XkbKTMapEntryPtr;
```

The *mods* field of a key type is an XkbModsRec (see section 7.2) specifying the modifiers the key type uses when calculating the shift level, and can be composed of both the core modifiers and virtual modifiers. To set the modifiers associated with a key type, modify the *real_mods* and *vmods* fields of the *mods* XkbModsRec accordingly. The *mask* field of the XkbModsRec is reserved for use by Xkb, and is calculated from the *real_mods* and *vmods* fields.

The *num_levels* field holds the total number of shift levels for the key type. Xkb uses *num_levels* to ensure the array of symbols bound to a key is large enough. Do not modify *num_levels* directly to change the number of shift levels for a key type. Instead, call *XkbResizeKeyType* (see section 15.2.3).

The *map* field is a vector of `XkbKtMapEntryRec` structures, with *map_count* entries, that specify the modifier combinations for each possible shift level. Each map entry contains an *active* field, a *mods* field, and a *level* field. The *active* field determines whether the modifier combination listed in the *mods* field should be considered when determining shift level. If *active* is False, this *map* entry is ignored. If *active* is True, the *level* field of the *map* entry specifies the shift level to use when the current modifier combination matches the combination specified in the *mods* field of the *map* entry.

Any combination of modifiers not explicitly listed somewhere in the *map* yields shift level one. In addition, *map* entries specifying unbound virtual modifiers are not considered.

Any modifiers specified in *mods* are normally *consumed* by `XkbTranslateKeyCode` (see section 12.1.3). For those rare occasions a modifier *should* be considered despite having been used to look up a symbol, key types include an optional *preserve* field. If a *preserve* member of a key type is not NULL, it represents a list of modifiers where each entry corresponds directly to one of the key type's *map*. Each entry lists the modifiers that should *not* be consumed if the matching map entry is used to determine shift level.

Each shift level has a name and these names are held in the *level_names* array, whose length is *num_levels*. The type itself also has a name, which is held in the *name* field.

For example, consider how the server handles the following possible symbolic description of a possible key type (Note that the format used to specify keyboard mappings in the server database is not specified by the Xkb extension, although this format is one possible example.):

Table 15.1 Example Key Type

<i>Symbolic Description</i>	<i>Key Type Data Structure</i>
type "ALPHATHREE" {	Xkb->map->types[i].name
modifiers = Shift+Lock+LevelThree;	Xkb->map->types[i].mods
map[None]= Level1;	Xkb->map->types[i].map[0]
map[Lock]= Level1;	Xkb->map->types[i].map[1]
map[Shift]= Level2;	Xkb->map->types[i].map[2]
map[LevelThree]= Level3;	Xkb->map->types[i].map[3]
map[Shift+LevelThree]= Level3;	Xkb->map->types[i].map[4]
preserve[None]= None;	Xkb->map->types[i].perserve[0]
preserve[Lock]= Lock;	Xkb->map->types[i].preserve[1]
preserve[Shift]= None;	Xkb->map->types[i].preserve[2]
preserve[LevelThree]= None;	Xkb->map->types[i].preserve[3]
preserve[Shift+Level3]= None;	Xkb->map->types[i].preserve[4]
level_name[Level1]= "Base";	Xkb->map->types[i].level_names[0]
level_name[Level2]= "Caps";	Xkb->map->types[i].level_names[1]
level_name[Level3]= "Level3";	Xkb->map->types[i].level_names[2]
};	

The *name* of the example key type is "ALPHATHREE," and the modifiers it pays attention to are Shift, Lock, and the virtual modifier LevelThree. There are three shift levels. The name of shift level one is "Base," the name of shift level two is "Caps," and the name of shift level three is "Level3."

Given the combination of the *map* and *preserve* specifications, there are five *map* entries. The first map entry specifies that shift level one is to be used if no modifiers are set. The second entry specifies the Lock modifier alone also yields shift level one. The third entry specifies the Shift modifier alone yields shift level two. The fourth and fifth entries specify that the virtual LevelThree modifier alone, or in combination with the Shift modifier, yields shift level three.

Note Shift level three can only be reached if the virtual modifier LevelThree is bound to a real modifier (see section 16.4). If LevelThree is not bound to a real modifier, the *map* entries associated with it are ignored.

Because the Lock modifier is to be preserved for further event processing, the *preserve* list is not NULL, and parallels the *map* list. All *preserve* entries, except for the one corresponding to the *map* entry that specifies the Lock modifier, do not list any modifiers. For the *map* entry that specifies the Lock modifier, the corresponding *preserve* list entry lists the Lock modifier, meaning do not consume the Lock modifier. In this particular case, the preserved modifier is passed to Xlib translation routines and causes them to notice that the Lock modifier is set; consequently, the Xlib routines apply the appropriate capitalization rules to the symbol. Since this preserve entry is set only for a modifier which yields shift level one, the capitalization occurs only for level one symbols.

15.2.1 The Canonical Key Types

Xkb allows up to XkbMaxKeyTypes (255) key types to be defined, but requires at least XkbNumRequiredTypes (4) predefined types to be in a key map. These predefined key types are referred to as the canonical key types and describe the types of keys available on most keyboards. The definitions for the canonical key types are held in the first XkbNumRequiredTypes entries of the *types* field of the client map, and are indexed using the following constants:

```
XkbOneLevelIndex
XkbTwoLevelIndex
XkbAlphabeticIndex
XkbKeypadIndex
```

ONE_LEVEL

The ONE_LEVEL key type describes groups that have only one symbol. The default ONE_LEVEL key type has no map entries and does not pay attention to any modifiers. A symbolic representation of this key type could look like the following:

```
type "ONE_LEVEL" {
    modifiers = None;
    map[None]= Level1;
    level_name[Level1]= "Any";
};
```

The description of the ONE_LEVEL key type is stored in the *types*[XkbOneLevelIndex] entry of the client key map.

TWO_LEVEL

The TWO_LEVEL key type describes groups that consist of two symbols but are neither alphabetic nor numeric keypad keys. The default TWO_LEVEL type uses only the Shift

modifier. It returns shift level two if Shift is set, and level one if it is not. A symbolic representation of this key type could look like the following:

```
type "TWO_LEVEL" {
    modifiers = Shift;
    map[Shift]= Level2;
    level_name[Level1]= "Base";
    level_name[Level2]= "Shift";
};
```

The description of the TWO_LEVEL key type is stored in the *types[XkbTwoLevelIndex]* entry of the client key map.

ALPHABETIC

The ALPHABETIC key type describes groups consisting 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 CapsLock” 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 so Xlib notices and applies the appropriate capitalization rules. The Xlib routines are locale-sensitive, and apply different capitalization rules for different locales.
- If neither Shift nor Lock is set, it yields level one.

A symbolic representation of this key type could look like the following:

```
type "ALPHABETIC" {
    modifiers = Shift+Lock;
    map[Shift]= Level2;
    preserve[Lock]= Lock;
    level_name[Level1]= "Base";
    level_name[Level2]= "Caps";
};
```

The description of the ALPHABETIC key type is stored in the *types[XkbAlphabeticIndex]* entry of the client key map.

KEYPAD

The KEYPAD key type describes groups that consist of two symbols, at least one of which is a numeric keypad symbol. The numeric keypad symbol is assumed to reside at level two. The default KEYPAD key type implements “Shift cancels NumLock” behavior using the Shift modifier and the real modifier bound to the virtual modifier named “NumLock,” known as the NumLock modifier, as follows:

- If Shift and NumLock are both set, the default KEYPAD type yields level one.
- If Shift alone is set, it yields level two.
- If NumLock alone is set, it yields level two.
- If neither Shift nor NumLock is set, it yields level one.

A symbolic representation of this key type could look like the following:

```
type "KEYPAD" {
    modifiers = Shift+NumLock;
    map[None]= Level1;
    map[Shift]= Level2;
    map[NumLock]= Level2;
    map[Shift+NumLock]= Level1;
    level_name[Level1]= "Base";
    level_name[Level2]= "Caps";
};
```

The description of the KEYPAD key type is stored in the `types[XkbKeypadIndex]` entry of the client key map.

Initializing the Canonical Key Types in a New Client Map

To set the definitions of the canonical key types in a client map to their default values, call `XkbInitCanonicalKeyTypes`.

Status `XkbInitCanonicalKeyTypes(xkb, which, keypadVMod)`

<code>XkbDescPtr</code>	<code>xkb;</code>	/* keyboard description containing client map to initialize */
<code>unsigned int</code>	<code>which;</code>	/* mask of types to initialize */
<code>int</code>	<code>keypadVMod;</code>	/* index of NumLock virtual modifier */

`XkbInitCanonicalKeyTypes` initializes the first `XkbNumRequiredTypes` key types of the keyboard specified by the `xkb` parameter to their default values. The `which` parameter specifies what canonical key types to initialize and is a bitwise inclusive OR of the following masks: `XkbOneLevelMask`, `XkbTwoLevelMask`, `XkbAlphabeticMask`, and `XkbKeypadMask`. Only those canonical types specified by the `which` mask are initialized.

If `XkbKeypadMask` is set in the `which` parameter, `XkbInitCanonicalKeyTypes` looks up the NumLock named virtual modifier to determine which virtual modifier to use when initializing the KEYPAD key type. If the NumLock virtual modifier does not exist, `XkbInitCanonicalKeyTypes` creates it.

`XkbInitCanonicalKeyTypes` normally returns Success. It returns BadAccess if the Xkb extension has not been properly initialized, and BadAccess if the `xkb` parameter is not valid.

15.2.2 Getting Key Types from the Server

To obtain the list of available key types in the server's keyboard mapping, call `XkbGetKeyTypes`.

Status `XkbGetKeyTypes(dpy, first, num, xkb)`

<code>Display *</code>	<code>dpy;</code>	/* connection to X server */
<code>unsigned int</code>	<code>first;</code>	/* index to first type to get, 0 => 1st type */
<code>unsigned int</code>	<code>num;</code>	/* number of key types to be returned */
<code>XkbDescPtr</code>	<code>xkb;</code>	/* keyboard description containing client map to update */

Note `XkbGetKeyTypes` is used to obtain descriptions of the key types themselves, not the key types bound to individual keys. To obtain the key types bound to an individual key, refer to the `key_sym_map` field of the client map (see section 15.3.1).

XkbGetKeyTypes queries the server for the desired types, waits for a reply, and returns the desired types in the *xkb->map->types*. If successful, it returns Success.

XkbGetKeyTypes returns BadAccess if the Xkb extension has not been properly initialized, and BadValue if the combination of *first* and *num* results in numbers out of valid range.

15.2.3 Changing the Number of Levels in a Key Type

To change the number of levels in a key type, call *XkbResizeKeyType*.

```
Status XkbResizeKeyType(xkb, type_ndx, map_count, want_preserve, new_num_lvls)
    XkbDescPtr xkb;           /* keyboard description containing client map to update */
    int         type_ndx;     /* index in xkb->map->types of type to change */
    int         map_count;    /* total # of map entries needed for the type */
    Bool        want_preserve; /* True => list of preserved modifiers is necessary */
    int         new_num_lvls; /* new max # of levels for type */
```

XkbResizeKeyType changes the type specified by *xkb->map->types[type_ndx]*, and reallocates the symbols and actions bound to all keys that use the type, if necessary. *XkbResizeKeyType* updates only the local copy of the types in *xkb*; to update the server's copy for the physical device, use *XkbSetMap* or *XkbChangeMap* after calling *XkbResizeKeyType*.

The *map_count* parameter specifies the total number of map entries needed for the type, and can be 0 or greater. If *map_count* is 0, *XkbResizeKeyType* frees the existing *map* and *preserve* entries for the type if they exist and sets them to NULL.

The *want_preserve* parameter specifies whether a *preserve* list for the key should be created. If *want_preserve* is True, the *preserve* list with *map_count* entries is allocated or reallocated if it already exists. Otherwise, if *want_preserve* is False, the *preserve* field is freed if necessary and set to NULL.

The *new_num_lvls* parameter specifies the new maximum number of shift levels for the type and is used to calculate and resize the symbols and actions bound to all keys that use the type.

If *type_ndx* does not specify a legal type, *new_num_lvls* is less than 1, or the *map_count* is less than 0, *XkbResizeKeyType* returns BadValue. If *XkbResizeKeyType* encounters any problems with allocation, it returns BadAlloc. Otherwise, it returns Success.

15.2.4 Copying Key Types

Use *XkbCopyKeyType* and *XkbCopyKeyTypes* to copy one or more *XkbKeyTypeRec* structures.

```
Status XkbCopyKeyType(from, into)
    XkbKeyTypePtr   from;          /* pointer to XkbKeyTypeRec to be copied */
    XkbKeyTypePtr   into;         /* pointer to XkbKeyTypeRec to be changed */
```

XkbCopyKeyType copies the key type specified by *from* to the key type specified by *into*. Both must point to legal *XkbKeyTypeRec* structures. Xkb assumes *from* and *into* point to different places. As a result, overlaps can be fatal. *XkbCopyKeyType* frees any existing *map*, *preserve*, and *level_names* in *into* prior to copying. If any allocation errors occur

while copying *from* to *into*, *XkbCopyKeyType* returns *BadAlloc*. Otherwise, *XkbCopyKeyType* copies *from* to *into* and returns *Success*.

Status ***XkbCopyKeyTypes***(*from*, *into*, *num_types*)

<i>XkbKeyTypePtr</i>	<i>from</i> ; /* pointer to array of <i>XkbKeyTypeRecs</i> to copy */
<i>XkbKeyTypePtr</i>	<i>into</i> ; /* pointer to array of <i>XkbKeyTypeRecs</i> to change */
<i>int</i>	<i>num_types</i> ; /* number of types to copy */

XkbCopyKeyTypes copies *num_types* *XkbKeyTypeRec* structures from the array specified by *from* into the array specified by *into*. It is intended for copying between, rather than within, keyboard descriptions, so it doesn't check for overlaps. The same rules that apply to the *from* and *into* parameters in *XkbCopyKeyType* apply to each entry of the *from* and *into* arrays of *XkbCopyKeyTypes*. If any allocation errors occur while copying *from* to *into*, *XkbCopyKeyTypes* returns *BadAlloc*. Otherwise, *XkbCopyKeyTypes* copies *from* to *into* and returns *Success*.

15.3 Key Symbol Map

The entire list of key symbols for the keyboard mapping is held in the *syms* field of the client map. Whereas the core keyboard mapping is a two-dimensional array of *KeySyms* whose rows are indexed by keycode, the *syms* field of *Xkb* is a linear list of *KeySyms* that needs to be indexed uniquely for each key. This section describes the key symbol map and the methods for determining the symbols bound to a key.

The reason the *syms* field is a linear list of *KeySyms* is to reduce the memory consumption associated with a keymap; because *Xkb* allows individual keys to have multiple shift levels and a different number of groups per key, a single two-dimensional array of *KeySyms* would potentially be very large and sparse. Instead, *Xkb* provides a small two-dimensional array of *KeySyms* for each key. To store all of these individual arrays, *Xkb* concatenates each array together in the *syms* field of the client map.

In order to determine which *KeySyms* in the *syms* field are associated with each keycode, the client map contains an array of key symbol mappings, held in the *key_sym_map* field. The *key_sym_map* field is an array of *XkbSymMapRec* structures indexed by keycode. The *key_sym_map* array has *min_key_code* unused entries at the start to allow direct indexing using a keycode. All keycodes falling between the minimum and maximum legal keycodes, inclusive, have *key_sym_map* arrays, whether or not any key actually yields that code. The *KeySymMapRec* structure is defined as follows:

```
#define XkbNumKbdGroups          4
#define XkbMaxKbdGroup           (XkbNumKbdGroups-1)

typedef struct {                                /* map to keysyms for a single keycode */
    unsigned char      kt_index[XkbNumKbdGroups]; /* key type index for each group */
    unsigned char      group_info;    /* # of groups and out of range group handling */
    unsigned char      width;        /* max # of shift levels for key */
    unsigned short     offset;       /* index to keysym table in syms array */
} XkbSymMapRec, *XkbSymMapPtr;
```

These fields are described in detail in the following sections.

15.3.1 Per-Key Key Type Indices

The *kt_index* array of the `XkbSymMapRec` structure contains the indices of the key types (see section 15.2) for each possible group of symbols associated with the key. To obtain the index of a key type or the pointer to a key type, Xkb provides the following macros, to access the key types:

Note The array of key types is of fixed width and is large enough to hold key types for the maximum legal number of groups (`XkbNumKbdGroups`, currently four); if a key has fewer than `XkbNumKbdGroups` groups, the extra key types are reported but ignored.

```
int XkbKeyTypeIndex(xkb, keycode, group)      /* macro */
XkbDescPtr      xkb;                      /* Xkb description of interest */
KeyCode         keycode;                   /* keycode of interest */
int             group;                    /* group index */
```

`XkbKeyTypeIndex` computes an index into the *types* vector of the client map in *xkb* from the given *keycode* and *group* index.

```
XkbKeyTypePtr XkbKeyType(xkb, keycode, group) /* macro */
XkbDescPtr      xkb;                      /* Xkb description of interest */
KeyCode         keycode;                   /* keycode of interest */
int             group;                    /* group index */
```

`XkbKeyType` returns a pointer to the key type in the *types* vector of the client map in *xkb* corresponding to the given *keycode* and *group* index.

15.3.2 Per-Key Group Information

The *group_info* field of an `XkbSymMapRec` is an encoded value containing the number of groups of symbols bound to the key as well as the specification of the treatment of out-of-range groups. 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`. To obtain the number of groups of symbols bound to the key, use `XkbKeyNumGroups`. To change the number of groups bound to a key, use `XkbChangeTypesOfKey` (see section 15.3.6). To obtain a mask that determines the treatment of out-of-range groups, use `XkbKeyGroupInfo` and `XkbOutOfRangeGroupInfo`.

The keyboard controls (see Chapter 10) contain a *groups_wrap* field specifying the handling of illegal groups on a global basis. That is, when the user performs an action causing the effective group to go out of the legal range, the *groups_wrap* field specifies how to normalize the effective keyboard group to a group which is legal for the keyboard as a whole, but there is no guarantee that the normalized group will be within the range of legal groups for any individual key. The per-key *group_info* field specifies how a key treats a legal effective group if the key does not have a type specified for the group of concern. For example, the `Enter` key usually has just one group defined. If the user performs an action causing the global keyboard group to change to `Group2`, the *group_info* field for the `Enter` key describes how to handle this situation.

Out-of-range groups for individual keys are mapped to a legal group using the same options as are used for the overall keyboard group. The particular type of mapping used is

controlled by the bits set in the *group_info* flag, as shown in Table 15.2. See section 10.7.1 on page 70 for more details on the normalization methods in this table.

Table 15.2 group_info Range Normalization

<i>Bits set in group_info</i>	<i>Normalization method</i>
XkbRedirectInRange	XkbRedirectInRange
XkbClampInRange	XkbClampInRange
none of the above	XkbWrapInRange

Xkb provides the following macros to access group information:

```
int XkbKeyNumGroups(xkb, keycode) /* macro */
    XkbDescPtr      xkb; /* Xkb description of interest */
    KeyCode         keycode; /* keycode of interest */
```

XkbKeyNumGroups returns the number of groups of symbols bound to the key corresponding to *keycode*.

```
unsigned char XkbKeyGroupInfo(xkb, keycode) /*macro*/
    XkbDescPtr      xkb; /* Xkb description of interest */
    KeyCode         keycode; /* keycode of interest */
```

XkbKeyGroupInfo returns the *group_info* field from the *XkbSymMapRec* structure associated with the key corresponding to *keycode*.

```
unsigned char XkbOutOfRangeGroupInfo(grp_inf/* macro */
    unsigned char     grp_inf; /* group_info field of XkbSymMapRec */)
```

XkbOutOfRangeGroupInfo returns only the out-of-range processing information from the *group_info* field of an *XkbSymMapRec* structure.

```
unsigned char XkbOutOfRangeGroupNumber(grp_inf/* macro */
    unsigned char     grp_inf; /* group_info field of XkbSymMapRec */)
```

XkbOutOfRangeGroupNumber returns the out-of-range group number, represented as a group index, from the *group_info* field of an *XkbSymMapRec* structure.

15.3.3 Key Width

The maximum number of shift levels for a type is also referred to as the width of a key type. The *width* field of the *key_sym_map* entry for a key contains the width of the widest type associated with the key. The *width* field cannot be explicitly changed; it is updated automatically whenever the symbols or set of types bound to a key are changed.

15.3.4 Offset in to the Symbol Map

The key width and number of groups associated with a key are used to form a small two-dimensional array of *KeySyms* for a key. This array may be different sizes for different keys. The array for a single key is stored as a linear list, in row-major order. The arrays for all of the keys are stored in the *syms* field of the client map. There is one row for each group associated with a key, and one column for each level. The index corresponding to a given group and shift level is computed as:

$$\text{idx} = \text{group_index} * \text{key_width} + \text{shift_level}$$

The *offset* field of the *key_sym_map* entry for a key is used to access the beginning of the array.

Xkb provides the following macros, for accessing the *width* and *offset* for individual keys, as well as macros for accessing the two-dimensional array of symbols bound to the key:

```
int XkbKeyGroupsWidth(xkb, keycode)          /* macro */
      XkbDescPtr    xkb;           /* Xkb description of interest */
      KeyCode       keycode;        /* keycode of interest */
```

XkbKeyGroupsWidth computes the maximum width associated with the key corresponding to *keycode*.

```
int XkbKeyGroupWidth(xkb, keycode, grp)      /* macro */
      XkbDescPtr    xkb;           /* Xkb description of interest */
      KeyCode       keycode;        /* keycode of interest */
      int           grp;           /* group of interest */
```

XkbKeyGroupWidth computes the width of the type associated with the group *grp* for the key corresponding to *keycode*.

```
int XkbKeySymsOffset(xkb, keycode)          /* macro */
      XkbDescPtr    xkb;           /* Xkb description of interest */
      KeyCode       keycode;        /* keycode of interest */
```

XkbKeySymsOffset returns the offset of the two-dimensional array of keysyms for the key corresponding to *keycode*.

```
int XkbKeyNumSyms(xkb, keycode)          /* macro */
      XkbDescPtr    xkb;           /* Xkb description of interest */
      KeyCode       keycode;        /* keycode of interest */
```

XkbKeyNumSyms returns the total number of keysyms for the key corresponding to *keycode*.

```
KeySym * XkbKeySymsPtr(xkb, keycode)        /* macro */
      XkbDescPtr    xkb;           /* Xkb description of interest */
      KeyCode       keycode;        /* keycode of interest */
```

XkbKeySymsPtr returns the pointer to the two-dimensional array of keysyms for the key corresponding to *keycode*.

```
KeySym XkbKeySymEntry(xkb, keycode, shift, grp)/* macro */
      XkbDescPtr    xkb;           /* Xkb description of interest */
      KeyCode       keycode;        /* keycode of interest */
      int           shift;          /* shift level of interest */
      int           grp;           /* group of interest */
```

XkbKeySymEntry returns the keysym corresponding to shift level *shift* and group *grp* from the two-dimensional array of keysyms for the key corresponding to *keycode*

15.3.5 Getting the Symbol Map for Keys from the Server

To obtain the symbols for a subset of the keys in a keyboard description, call *XkbGetKeySyms*:

```
Status XkbGetKeySyms(dpy, first, num, xkb)
    Display *          dpy;           /* connection to X server */
    unsigned int       first;        /* keycode of first key to get */
    unsigned int       num;         /* number of keycodes for which syms desired */
    XkbDescPtr        xkb;         /* Xkb description to be updated */
```

XkbGetKeySyms sends a request to the server to obtain the set of keysyms bound to *num* keys starting with the key whose keycode is *first*. It waits for a reply, and returns the keysyms in the *map.syms* field of *xkb*. If successful, *XkbGetKeySyms* returns Success. The *xkb* parameter must be a pointer to a valid Xkb keyboard description.

If the client *map* in the *xkb* parameter has not been allocated, *XkbGetKeySyms* allocates and initializes it before obtaining the symbols.

If a compatible version of Xkb is not available in the server or the Xkb extension has not been properly initialized, *XkbGetKeySyms* returns BadAccess. If *num* is less than 1 or greater than *XkbMaxKeyCount*, *XkbGetKeySyms* returns BadValue. If any allocation errors occur, *XkbGetKeySyms* returns BadAlloc.

15.3.6 Changing the Number of Groups and Types Bound to a Key

To change the number of groups and the types bound to a key, call *XkbChangeTypesOfKey*.

```
Status XkbChangeTypesOfKey(xkb, key, n_groups, groups, new_types_in, p_changes)
```

XkbDescPtr	<i>xkb</i> ;	/* keyboard description to be changed */
int	<i>key</i> ;	/* keycode for key of interest */
int	<i>n_groups</i> ;	/* new number of groups for key */
unsigned int	<i>groups</i> ;	/* mask indicating groups to change */
int *	<i>new_types_in</i> ;	/* indices for new groups specified in <i>groups</i> */
XkbMapChangesPtr	<i>p_changes</i> ;	/* notes changes made to <i>xkb</i> */

XkbChangeTypesOfKey reallocates the symbols and actions bound to the key, if necessary, and initializes any new symbols or actions to NoSymbol or NoAction, as appropriate. If the *p_changes* parameter is not NULL, *XkbChangeTypesOfKey* adds the *XkbKeySymsMask* to the *changes* field of *p_changes*, and modifies the *first_key_sym* and *num_key_syms* fields of *p_changes* to include the *key* that was changed. See section 14.3.1 on page 120 for more information on the *XkbMapChangesPtr* structure. If successful, *XkbChangeTypesOfKey* returns Success.

The *n_groups* parameter specifies the new number of groups for the key. The *groups* parameter is a mask specifying the groups for which new types are supplied and is a bit-wise inclusive OR of the following masks: *XkbGroup1Mask*, *XkbGroup2Mask*, *XkbGroup3Mask*, and *XkbGroup4Mask*.

The *new_types_in* parameter is an integer array of length *n_groups*. Each entry represents the type to use for the associated group and is an index into *xkb->map->types*. The *new_types_in* array is indexed by group index; if *n_groups* is four and *groups* only has *Group1Mask* and *Group3Mask* set, *new_types_in* looks like this:

```

new_types_in[0] = type for Group1
new_types_in[1] = ignored
new_types_in[2] = type for Group3
new_types_in[3] = ignored

```

For convenience, Xkb provides the following constants to use as indices to the groups:

Table 15.3 Group Index Constants

Constant Name	Value
XkbGroup1Index	0
XkbGroup2Index	1
XkbGroup3Index	2
XkbGroup4Index	3

If the Xkb extension has not been properly initialized, *XkbChangeTypesOfKey* returns *BadAccess*. If the *xkb* parameter is not valid (that is, it is *NULL* or it does not contain a valid client map), *XkbChangeTypesOfKey* returns *BadMatch*. If the *key* is not a valid key-code, *n_groups* is greater than *XkbNumKbdGroups*, or the *groups* mask does not contain any of the valid group mask bits, *XkbChangeTypesOfKey* returns *BadValue*. If it is necessary to resize the key symbols or key actions arrays and any allocation errors occur, *XkbChangeTypesOfKey* returns *BadAlloc*.

15.3.7 Changing the Number of Symbols Bound to a Key

To change the number of symbols bound to a key, call *XkbResizeKeySyms*.

*KeySym *XkbResizeKeySyms(xkb, key, needed)*

<i>XkbDescRec *</i>	<i>xkb;</i>	/* keyboard description to be changed */
<i>int</i>	<i>key;</i>	/* keycode for key to modify */
<i>int</i>	<i>needed;</i>	/* new number of keysyms required for key */

XkbResizeKeySyms reserves the space needed for *needed* keysyms and returns a pointer to the beginning of the new array that holds the keysyms. It adjusts the *offset* field of the *key_sym_map* entry for the key if necessary, and can also change the *syms*, *num_syms*, and *size_syms* fields of *xkb->map* if it is necessary to reallocate the *syms* array. *XkbResizeKeySyms* does not modify either the width or number of groups associated with the key.

If *needed* is greater than the current number of keysyms for the key, *XkbResizeKeySyms* initializes all new keysyms in the array to *NoSymbol*.

Since the number of symbols needed by a key is normally computed as width * number of groups, and *XkbResizeKeySyms* does not modify either the width or number of groups for the key, a discrepancy exists upon return from *XkbResizeKeySyms* between the space allocated for the keysyms and the number required. The unused entries in the list of symbols returned by *XkbResizeKeySyms* are not preserved across future calls to any of the map editing functions, so you must update the key symbol mapping (which updates the width and number of groups for the key) before calling another allocator function. A call to *XkbChangeTypesOfKey* will update the mapping.

If any allocation errors occur while resizing the number of symbols bound to the key, *XkbResizeKeySyms* returns *NULL*.

Note A change to the number of symbols bound to a key should be accompanied by a change in the number of actions bound to a key. Refer to section 16.1.16 on page 160 for more information on changing the number of actions bound to a key.

15.4 The Per-Key Modifier Map

The *modmap* entry of the client map is an array, indexed by keycode, specifying the real modifiers bound to a key. Each entry is a mask composed of a bitwise inclusive OR of the legal real modifiers: ShiftMask, LockMask, ControlMask, Mod1Mask, Mod2Mask, Mod3Mask, Mod4Mask, and Mod5Mask. If a bit is set in a *modmap* entry, the corresponding key is bound to that modifier.

Pressing or releasing the key bound to a modifier changes the modifier set and unset state. The particular manner in which the modifier set and unset state changes is determined by the behavior and actions assigned to the key (see Chapter 16).

15.4.1 Getting the Per-Key Modifier Map from the Server

To update the modifier map for one or more of the keys in a keyboard description, call *XkbGetKeyModifierMap*.

```
Status XkbGetKeyModifierMap(dpy, first, num, xkb)
    Display *          dpy;           /* connection to X server */
    unsigned int       first;         /* keycode of first key to get */
    unsigned int       num;          /* number of keys for which information is desired */
    XkbDescPtr        xkb;          /* keyboard description to update */
```

XkbGetKeyModifierMap sends a request to the server for the modifier mappings for *num* keys starting with the key whose keycode is *first*. It waits for a reply, and places the results in the *xkb->map->modmap* array. If successful, *XkbGetKeyModifier* returns Success.

If the map component of the *xkb* parameter has not been allocated, *XkbGetKeyModifier-Map* allocates and initializes it.

If a compatible version of Xkb is not available in the server or the Xkb extension has not been properly initialized, *XkbGetKeySyms* returns BadAccess. If any allocation errors occur while obtaining the modifier map, *XkbGetKeyModifierMap* returns BadAlloc.

16 Xkb Server Keyboard Mapping

The *server* field of the complete Xkb keyboard description (see section 6.1) is a pointer to the Xkb server map.

Figure 16.1 shows the relationships between elements in the server map:

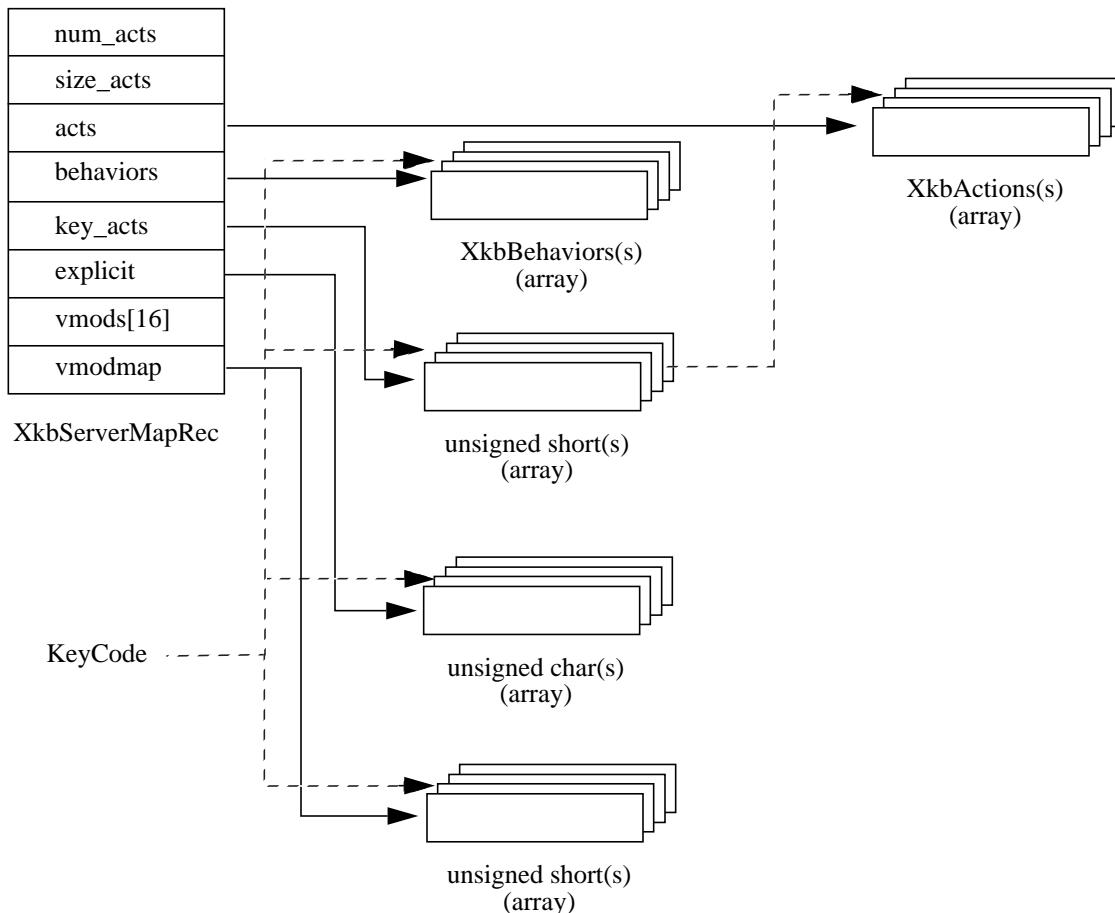


Figure 16.1 Server Map Relationships

The Xkb server map contains the information the server needs to interpret key events, and is of type `XkbServerMapRec`:

```

#define XkbNumVirtualMods      16

typedef struct {
    unsigned short      num_acts;          /* Server Map */
    unsigned short      size_acts;         /* # of occupied entries in acts */
    XkbAction *         acts;             /* linear 2d tables of key actions, 1 per keycode */
    XkbBehavior *       behaviors;        /* key behaviors, 1 per keycode */
    unsigned short *    key_acts;          /* index into acts, 1 per keycode */
    unsigned char *     explicit;         /* explicit overrides of core remapping, 1 per key */
    unsigned char       vmods[XkbNumVirtualMods]; /* real mods bound to virtual mods */

```

```
    unsigned short *    vmodmap;      /* virtual mods bound to key, 1 per keycode*/
} XkbServerMapRec, *XkbServerMapPtr;
```

The *num_acts*, *size_acts*, *acts*, and *key_acts* fields specify the key actions, defined in section 16.1. The *behaviors* field describes the behavior for each key and is defined in section 16.2. The *explicit* field describes the explicit components for a key and is defined in section 16.3. The *vmods* and the *vmodmap* fields describe the virtual modifiers and the per-key virtual modifier mapping and are defined in section 16.4.

16.1 Key Actions

A key action defines the effect key presses and releases have on the internal state of the server. For example, the expected key action associated with pressing the Shift key is to set the Shift modifier. There is zero or one key action associated with each keysym bound to each key.

Just as the entire list of key symbols for the keyboard mapping is held in the *syms* field of the client map, the entire list of key actions for the keyboard mapping is held in the *acts* array of the server map. The total size of *acts* is specified by *size_acts*, and the number of entries is specified by *num_acts*.

The *key_acts* array, indexed by keycode, describes the actions associated with a key. The *key_acts* array has *min_key_code* unused entries at the start to allow direct indexing using a keycode. If a *key_acts* entry is 0, it means the key does not have any actions associated with it. If an entry is not 0, the entry represents an index into the *acts* field of the server map, much as the *offset* field of a KeySymMapRec structure is an index into the *syms* field of the client map.

The reason the *acts* field is a linear list of XkbActions is to reduce the memory consumption associated with a keymap. Because Xkb allows individual keys to have multiple shift levels and a different number of groups per key, a single two-dimensional array of KeySyms would potentially be very large and sparse. Instead, Xkb provides a small two-dimensional array of XkbActions for each key. To store all of these individual arrays, Xkb concatenates each array together in the *acts* field of the server map.

The key action structures consist only of fields of type char or unsigned char. This is done to optimize data transfer when the server sends bytes over the wire. If the fields are anything but bytes, the server has to sift through all of the actions and swap any non-byte fields. Since they consist of nothing but bytes, it can just copy them out.

Xkb provides the following macros, to simplify accessing information pertaining to key actions:

Bool XkbKeyHasActions (<i>xkb</i> , <i>keycode</i>)	/* macro */
XkbDescPtr <i>xkb</i> ;	/* Xkb description of interest */
KeyCode <i>keycode</i> ;	/* keycode of interest */

XkbKeyHasActions returns True if the key corresponding to *keycode* has any actions associated with it; otherwise, it returns False.

int XkbKeyNumActions (<i>xkb</i> , <i>keycode</i>)	/* macro */
XkbDescPtr <i>xkb</i> ;	/* Xkb description of interest */
KeyCode <i>keycode</i> ;	/* keycode of interest */

XkbKeyNumActions computes the number of actions associated with the key corresponding to *keycode*. This should be the same value as the result of *XkbKeyNumSyms* (see section 15.3.3).

```
XkbKeyActionPtr XkbKeyActionsPtr(xkb, keycode)/* macro */
    XkbDescPtr      xkb;           /* Xkb description of interest */
    KeyCode         keycode;        /* keycode of interest */
```

XkbKeyActionsPtr returns a pointer to the two-dimensional array of key actions associated with the key corresponding to *keycode*. Use *XkbKeyActionsPtr* only if the key actually has some actions associated with it, that is, *XkbKeyNumActions(xkb, keycode)* returns something greater than zero.

```
XkbAction XkbKeyAction(xkb, keycode, idx)      /* macro */
    XkbDescPtr      xkb;           /* Xkb description of interest */
    KeyCode         keycode;        /* keycode of interest */
    int             idx;           /* index for group and shift level */
```

XkbKeyAction returns the key action indexed by *idx* in the two-dimensional array of key actions associated with the key corresponding to *keycode*. *idx* may be computed from the group and shift level of interest as follows:

$$\text{idx} = \text{group_index} * \text{key_width} + \text{shift_level}$$

```
XkbAction XkbKeyActionEntry(xkb, keycode, shift, grp)/* macro */
    XkbDescPtr      xkb;           /* Xkb description of interest */
    KeyCode         keycode;        /* keycode of interest */
    int             shift;          /* shift level within group */
    int             grp;           /* group index for group of interest */
```

XkbKeyActionEntry returns the key action corresponding to group *grp* and shift level *lvl* from the two-dimensional table of key actions associated with the key corresponding to *keycode*.

16.1.1 The XkbAction Structure

The description for an action is held in an *XkbAction* structure, which is a union of all possible Xkb action types:

```
typedef union _XkbAction {
    XkbAnyAction      any;
    XkbModAction     mods;
    XkbGroupAction   group;
    XkbISOAction     iso;
    XkbPtrAction     ptr;
    XkbPtrBtnAction  btn;
    XkbPtrDfltAction dflt;
    XkbSwitchScreenAction screen;
    XkbCtrlsAction   ctrls;
    XkbMessageAction msg;
    XkbRedirectKeyAction redirect;
    XkbDeviceBtnAction devbtn;
    XkbDeviceValuatorAction devval;
```

```
    unsigned char      type;
} XkbAction;
```

The *type* field is provided for convenience and is the same as the *type* field in the individual structures. The following sections describe the individual structures for each action in detail.

16.1.2 The XkbAnyAction Structure

The XkbAnyAction structure is a convenience structure that refers to any of the actions:

```
#define XkbAnyActionDataSize      7
typedef struct _XkbAnyAction {
    unsigned char      type;          /* type of action; determines interpretation for data */
    unsigned char      data[XkbAnyActionDataSize];
} XkbAnyAction;
```

The *data* field represents a structure for an action, and its interpretation depends upon the *type* field. The valid values for the *type* field, and the data structures associated with them are shown in Table 16.1:

Table 16.1 Action Types

Type	Structure for Data	XkbAction union member	Section
XkbSA_NoAction	XkbSA_NoAction means the server does not perform an action for the key; this action does not have an associated data structure.	any	
XkbSA_SetMods	XkbModAction	mods	16.1.3
XkbSA_LatchMods			
XkbSA_LockMods			
XkbSA_SetGroup	XkbGroupAction	group	16.1.4
XkbSA_LatchGroup			
XkbSA_LockGroup			
XkbSA_MovePtr	XkbPtrAction	ptr	16.1.5
XkbSA_PtrBtn	XkbPtrBtnAction	btn	16.1.6
XkbSA_LockPtrBtn			
XkbSA_SetPtrDflt	XkbPtrDfltAction	dflt	16.1.7
XkbSA_ISOLock	XkbISOAction	iso	16.1.8
XkbSA_SwitchScreen	XkbSwitchScreenAction	screen	16.1.9
XkbSA_SetControls	XkbCtrlsAction	ctrls	16.1.10
XkbSA_LockControls			
XkbSA_ActionMessage	XkbMessgeAction	msg	16.1.11
XkbSA_RedirectKey	XkbRedirectKeyAction	redirect	16.1.12
XkbSA_DeviceBtn	XkbDeviceBtnAction	devbtn	16.1.13
XkbSA_LockDeviceBtn			
XkbSA_DeviceValuator	XkbDeviceValuatorAction	devval	16.1.14

16.1.3 Actions for Changing Modifiers' State

Actions associated with the XkbModAction structure change the state of the modifiers when keys are pressed and released (see Chapter 7 for a discussion of modifiers):

```

typedef struct _XkbModAction {
    unsigned char type;      /* XkbSA_{Set|Latch|Lock}Mods */
    unsigned char flags;     /* with type, controls the effect on modifiers */
    unsigned char mask;      /* same as mask field of a modifier description */
    unsigned char real_mods; /* same as real_mods field of a modifier description */
    unsigned char vmods1;    /* derived from vmods field of a modifier description */
    unsigned char vmods2;    /* derived from vmods field of a modifier description */
} XkbModAction;

```

In the following description, the term *action modifiers* means the real modifier bits associated with this action. Depending on the value of *flags* (see Table 16.3 on page 145), these are designated either in the *mask* field of the `XkbModAction` structure itself, or the real modifiers bound to the key for which the action is being used. In the latter case, this is the client `map->modmap[keycode]` field.

The *type* field can have any of the values shown in Table 16.2.

Table 16.2 Modifier Action Types

Type	Effect
XkbSA_SetMods	<ul style="list-style-type: none"> A key press adds any action modifiers to the keyboard's base modifiers. A key release clears any action modifiers in the keyboard's base modifiers, provided no other key affecting the same modifiers is logically down. If no other keys are physically depressed when this key is released, and <code>XkbSA_ClearLocks</code> is set in the <i>flags</i> field, the key release unlocks any action modifiers.
XkbSA_LatchMods	<ul style="list-style-type: none"> Key press and key release events have the same effect as for <code>XkbSA_SetMods</code>; if no keys are physically depressed when this key is released, key release events have the following additional effects: <ul style="list-style-type: none"> Modifiers unlocked due to <code>XkbSA_ClearLocks</code> have no further effect. If <code>XkbSA_LatchToLock</code> is set in the <i>flags</i> field, a key release locks and then unlatches any remaining action modifiers that are already latched. A key release latches any action modifiers not used by the <code>XkbSA_ClearLocks</code> and <code>XkbSA_LatchToLock</code> flags.
XkbSA_LockMods	<ul style="list-style-type: none"> A key press sets the base state of any action modifiers. If <code>XkbSA_LockNoLock</code> is set in the <i>flags</i> field, a key press also sets the locked state of any action modifiers. A key release clears any action modifiers in the keyboard's base modifiers, provided no other key that affects the same modifiers is down. If <code>XkbSA_LockNoUnlock</code> is not set in the <i>flags</i> field, and any of the action modifiers were locked before the corresponding key press occurred, a key release unlocks them.

The *flags* field is composed of the bitwise inclusive OR of the masks shown in Table 16.3. A general meaning is given in the table, but the exact meaning depends upon the action type.

Table 16.3 Modifier Action Flags

Flag	Meaning
XkbSA_UseModMapMods	If set, the action modifiers are determined by the modifiers bound by the modifier mapping of the key. Otherwise, the action modifiers are set to the modifiers specified by the <i>mask</i> , <i>real_mods</i> , <i>vmod1</i> , and <i>vmod2</i> fields.
XkbSA_ClearLocks	If set and no keys are physically depressed when this key transition occurs, the server unlocks any action modifiers.
XkbSA_LatchToLock	If set, and the action type is XkbSA_LatchMods, the server locks the action modifiers if they are already latched.
XkbSA_LockNoLock	If set, and the action type is XkbSA_LockMods, the server only unlocks the action modifiers.
XkbSA_LockNoUnlock	If set, and the action is XkbSA_LockMods, the server only locks the action modifiers.

If XkbSA_UseModMapMods is not set in the *flags* field, the *mask*, *real_mods*, *vmod1*, and *vmod2* fields are used to determine the action modifiers. Otherwise they are ignored and the modifiers bound to the key (client *map->modmap[keycode]*) are used instead.

The *mask*, *real_mods*, *vmod1*, and *vmod2* fields represent the components of an Xkb modifier description (see section 7.2). While the *mask* and *real_mods* fields correspond directly to the *mask* and *real_mods* fields of an Xkb modifier description, the *vmod1* and *vmod2* fields are combined to correspond to the *vmods* field of an Xkb modifier description. Xkb provides the following macros, to convert between the two formats:

```
unsigned short XkbModActionVMods(act)      /* macro */
XkbAction          act;                  /* action from which to extract virtual mods */
```

XkbModActionVMods returns the *vmod1* and *vmod2* fields of *act* converted to the *vmods* format of an Xkb modifier description.

```
void XkbSetModActionVMods(act, vmods)    /* macro */
XkbAction          act;                  /* action in which to set vmods */
unsigned short     vmods;                /* virtual mods to set */
```

XkbSetModActionVMods sets the *vmod1* and *vmod2* fields of *act* using the *vmods* format of an Xkb modifier description.

Note Despite the fact that the first parameter of these two macros is of type XkbAction, these macros may only be used with Actions of type XkbModAction and XkbISOAction.

16.1.4 Actions for Changing Group State

Actions associated with the XkbGroupAction structure change the current group state when keys are pressed and released (see Chapter 5 for a description of groups and keyboard state):

```
typedef struct _XkbGroupAction {
    unsigned char      type;           /* XkbSA_{Set|Latch|Lock}Group */
    unsigned char      flags;          /* with type, controls the effect on groups */
```

```
char group_XXX; /* represents a group index or delta */
} XkbGroupAction;
```

The *type* field can have any of the following values:

Table 16.4 Group Action Types

Type	Effect
XkbSA_SetGroup	<ul style="list-style-type: none"> If the XkbSA_GroupAbsolute bit is set in the <i>flags</i> field, key press events change the base keyboard group to the group specified by the <i>group_XXX</i> field. Otherwise key press events change the base keyboard group by adding the <i>group_XXX</i> field to the base keyboard group. In either case, the resulting effective keyboard group is brought back into range depending upon the value of the <i>groups_wrap</i> field of the controls structure (see section 10.7.1). If a key with an XkbSA_ISOLock action (see section 16.1.8) is pressed while this key is down, the key release of this key has no effect. Otherwise, the key release cancels the effects of the key press. If the XkbSA_ClearLocks bit is set in the <i>flags</i> field, and no keys are physically depressed when this key is released, the key release also sets the locked keyboard group to Group1.
XkbSA_LatchGroup	<ul style="list-style-type: none"> Key press and key release events have the same effect as for XkbSA_SetGroup; if no keys are physically depressed when this key is released, key release events have the following additional effects: If the XkbSA_LatchToLock bit is set in the <i>flags</i> field and the latched keyboard group index 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 <i>groups_wrap</i> field of the controls structure. Otherwise, the key press adds the key press delta to the latched keyboard group.
XkbSA_LockGroup	<ul style="list-style-type: none"> If the XkbSA_GroupAbsolute is set in the <i>flags</i> field, key press events set the locked keyboard group to the group specified by the <i>group_XXX</i> field. Otherwise, key press events add the group specified by the <i>group_XXX</i> field to the locked keyboard group. In either case, the resulting locked and effective keyboard groups are brought back into range depending upon the value of the <i>groups_wrap</i> field of the controls structure A key release has no effect.

The *flags* field is composed of the bitwise inclusive OR of the masks shown in Table 16.5. A general meaning is given in the table, but the exact meaning depends upon the action type.

Table 16.5 Group Action Flags

Flag	Meaning
XkbSA_ClearLocks	If set and no keys are physically depressed when this key transition occurs, the server sets the locked keyboard group to Group1 on a key release.
XkbSA_LatchToLock	If set, and the action type is SA_LatchGroup, the server locks the action group if it is already latched.
XkbSA_GroupAbsolute	If set, the <i>group_XXX</i> field represents an absolute group number. Otherwise, it represents a group delta to be added to the current group to determine the new group number.

The *group_XXX* field represents a signed character. Xkb provides the following macros, to convert between a signed integer value and a signed character:

```
int XkbSAGroup(act)          /* macro */
    XkbAction      act;        /* action from which to extract group */
```

XkbSAGroup returns the *group_XXX* field of *act* converted to a signed int.

```
void XkbSASetGroup(act, grp)   /* macro */
    XkbAction      act;        /* action from which to set group */
    int            grp;        /* group index to set in group_XXX */
```

XkbSASetGroup sets the *group_XXX* field of *act* from the group index *grp*.

Note Despite the fact that the first parameter of these two macros is of type XkbAction, these macros may only be used with Actions of type XkbGroupAction and XkbISOACTION.

16.1.5 Actions for Moving the Pointer

Actions associated with the XkbPtrAction structure move the pointer when keys are pressed and released:

```
typedef struct _XkbPtrAction {
    unsigned char    type;      /* XkbSA_MovePtr */
    unsigned char    flags;     /* determines type of pointer motion */
    unsigned char    high_XXX;  /* x co-ordinate, high bits */
    unsigned char    low_XXX;   /* y co-ordinate, low bits */
    unsigned char    high_YYY;  /* x co-ordinate, high bits */
    unsigned char    low_YYY;   /* y co-ordinate, low bits */
} XkbPtrAction;
```

If the MouseKeys control is not enabled (see section 10.5.1), KeyPress and KeyRelease events are treated as though the action is XkbSA_NoAction.

If the MouseKeys control is enabled, a server action of type XkbSA_MovePtr instructs the server to generate core pointer MotionNotify events rather than the usual KeyPress event, and the corresponding KeyRelease event disables any mouse keys timers that were created as a result of handling the XkbSA_MovePtr action.

The *type* field of the XkbPtrAction structure is always XkbSA_MovePtr.

The *flags* field is a bitwise inclusive OR of the masks shown in Table 16.6.

Table 16.6 Pointer Action Types

Action Type	Meaning
XkbSA_NoAcceleration	If not set, and the MouseKeysAccel control is enabled (see section 10.5.2), the KeyPress initiates a mouse keys timer for this key; every time the timer expires, the cursor moves.
XkbSA_MoveAbsoluteX	If set, the X portion of the structure specifies the new pointer X coordinate. Otherwise, the X portion is added to the current pointer X coordinate to determine the new pointer X coordinate.

Table 16.6 Pointer Action Types

Action Type	Meaning
XkbSA_MoveAbsoluteY	If set, the Y portion of the structure specifies the new pointer Y coordinate. Otherwise, the Y portion is added to the current pointer Y coordinate to determine the new pointer Y coordinate.

Each of the X and Y co-ordinates of the XkbPtrAction structure is composed of two signed 16-bit values, that is, the X co-ordinate is composed of *high_XXX* and *low_XXX*, and similarly for the Y co-ordinate. Xkb provides the following macros, to convert between a signed integer and two signed 16-bit values in XkbPtrAction structures:

```
int XkbPtrActionX(act)          /* macro */
    XkbPtrAction      act;      /* action from which to extract X */
```

XkbPtrActionX returns the *high_XXX* and *low_XXX* fields of *act* converted to a signed int.

```
int XkbPtrActionY(act)          /* macro */
    XkbPtrAction      act;      /* action from which to extract Y */
```

XkbPtrActionY returns the *high_YYY* and *low_YYY* fields of *act* converted to a signed int.

```
void XkbSetPtrActionX(act, x)   /* macro */
    XkbPtrAction      act;      /* action in which to set X */
    int               x;        /* new value to set */
```

XkbSetPtrActionX sets the *high_XXX* and *low_XXX* fields of *act* from the signed integer value *x*.

```
void XkbSetPtrActionY(act, y)   /* macro */
    XkbPtrAction      act;      /* action in which to set Y */
    int               y;        /* new value to set */
```

XkbSetPtrActionY sets the *high_YYY* and *low_YYY* fields of *act* from the signed integer value *y*.

16.1.6 Actions for Simulating Pointer Button Press and Release

Actions associated with the XkbPtrBtnAction structure simulate the press and release of pointer buttons when keys are pressed and released:

```
typedef struct _XkbPtrBtnAction {
    unsigned char  type;  /*XkbSA_PtrBtn, XkbSA_LockPtrBtn */
    unsigned char  flags; /* with type, controls the effect on pointer buttons*/
    unsigned char  count; /* controls number of ButtonPress and ButtonRelease events */
    unsigned char  button; /* pointer button to simulate */
} XkbPtrBtnAction;
```

If the MouseKeys (see section 10.5.1) control is not enabled, KeyPress and KeyRelease events are treated as though the action is XkbSA_NoAction.

The *type* field can have any one of the values shown in Table 16.7.

Table 16.7 Pointer Button Action Types

Type	Effect
XkbSA_PtrBtn	<ul style="list-style-type: none"> If <code>XkbSA_UseDfltButton</code> is set in the <i>flags</i> field, the event is generated for the pointer button specified by the <code>mk_dflt_btn</code> attribute of the <code>MouseKeys</code> control (see section 10.5.1). Otherwise, the event is generated for the button specified by the <i>button</i> field. If the mouse button specified for this action is logically down, the key press and corresponding key release are ignored and have no effect. Otherwise, a key press causes one or more core pointer button events instead of the usual <code>KeyPress</code> event. If <i>count</i> is 0, a key press generates a single <code>ButtonPress</code> event; if <i>count</i> is greater than 0, a key press generates <i>count</i> pairs of <code>ButtonPress</code> and <code>ButtonRelease</code> events. If <i>count</i> is 0, a key release generates a core pointer <code>ButtonRelease</code> which matches the event generated by the corresponding <code>KeyPress</code>; if <i>count</i> is non-zero, a key release does not cause a <code>ButtonRelease</code> event. A key release never generates a key <code>KeyRelease</code> event.
XkbSA_LockPtrBtn	<ul style="list-style-type: none"> If the button specified by the <code>MouseKeys</code> default button or <i>button</i> is not locked, a key press causes a <code>ButtonPress</code> event instead of a <code>KeyPress</code> event and locks the button. If the button is already locked or if <code>XkbSA_LockNoUnlock</code> is set in the <i>flags</i> field, a key press is ignored and has no effect. If the corresponding key press was ignored, and if <code>XkbSA_LockNoLock</code> is not set in the <i>flags</i> field, a key release generates a <code>ButtonRelease</code> event instead of a <code>KeyRelease</code> event and unlocks the specified button. If the corresponding key press locked a button, the key release is ignored and has no effect.

The *flags* field is composed of the bitwise inclusive OR of the masks shown in Table 16.8. A general meaning is given in the table, but the exact meaning depends upon the action *type*:

Table 16.8 Pointer Button Action Flags

Flag	Meaning
<code>XkbSA_UseDfltButton</code>	If set, the action uses the pointer button specified by the <code>mk_dflt_btn</code> attribute of the <code>MouseKeys</code> control (see section 10.5.1). Otherwise, the action uses the pointer button specified by the <i>button</i> field.
<code>XkbSA_LockNoLock</code>	If set, and the action type is <code>XkbSA_LockPtrBtn</code> , the server only unlocks the pointer button.
<code>XkbSA_LockNoUnlock</code>	If set, and the action type is <code>XkbSA_LockPtrBtn</code> , the server only locks the pointer button.

16.1.7 Actions for Changing the Pointer Button Simulated

Actions associated with the `XkbPtrDfltAction` structure change the `mk_dflt_btn` attribute of the `MouseKeys` control (see section 10.5.1 on page 60):

```
typedef struct _XkbPtrDfltAction {
    unsigned char    type;          /* XkbSA_SetPtrDflt */
    unsigned char    flags;         /* controls the pointer button number */
    unsigned char    affect;        /* XkbSA_AffectDfltBtn */
```

```
char valueXXX; /* new default button member */
} XkbPtrDfltAction;
```

If the MouseKeys control is not enabled, KeyPress and KeyRelease events are treated as though the action is XkbSA_NoAction. Otherwise, this action changes the *mk_dflt_btn* attribute of the MouseKeys control.

The *type* field of the XkbPtrDfltAction structure should always be XkbSA_SetPtrDflt.

The *flags* field is composed of the bitwise inclusive OR of the values shown in Table 16.9 (currently there is only one value defined).

Table 16.9 Pointer Default Flags

Flag	Meaning
XkbSA_DfltBtnAbsolute	If set, the <i>value</i> field represents an absolute pointer button. Otherwise, the <i>value</i> field represents the amount to be added to the current default button.

The *affect* field specifies what changes as a result of this action. The only valid value for the *affect* field is XkbSA_AffectDfltBtn.

The *valueXXX* field is a signed character that represents the new button value for the *mk_dflt_btn* attribute of the MouseKeys control (see section 10.5.1). If XkbSA_DfltBtnAbsolute is set in *flags*, *valueXXX* specifies the button to be used; otherwise, *valueXXX* specifies the amount to be added to the current default button. In either case, illegal button choices are wrapped back around into range. Xkb provides the following macros, to convert between the integer and signed character values in XkbPtrDfltAction structures:

```
int XkbSAPtrDfltValue(act) /* macro */
XkbAction act; /* action from which to extract group */
```

XkbSAPtrDfltValue returns the *valueXXX* field of *act* converted to a signed int.

```
void XkbSASetPtrDfltValue(act, val) /* macro */
XkbPtrDfltAction act; /* action in which to set valueXXX */
int val; /* value to set in valueXXX */
```

XkbSASetPtrDfltValue sets the *valueXXX* field of *act* from *val*.

16.1.8 Actions for Locking Modifiers and Group

Actions associated with the XkbISOAction structure lock modifiers and the group according to the ISO9995 specification.

Operated by itself, the XkbISOAction is just a caps lock. Operated simultaneously with another modifier key, it transforms the other key into a locking key. For example, press ISO_Lock, press and release Control_L, release ISO_Lock ends up locking the Control modifier.

The default behavior is to convert:

```
{Set,Latch}Mods to: LockMods
{Set,Latch}Group to: LockGroup
SetPtrBtn to: LockPtrBtn
```

SetControls to: LockControls

The *affects* field allows you to turn those effects on or off individually. Set `XkbSA_ISONoAffectMods` to disable the first, `XkbSA_ISONoAffectGroup` to disable the second, and so forth.

```
typedef struct _XkbISOAction {
    unsigned char type;      /* XkbSA_ISOLock */
    unsigned char flags;     /* controls changes to group or modifier state */
    unsigned char mask;      /* same as mask field of a modifier description */
    unsigned char real_mods; /* same as real_mods field of a modifier description */
    char          group_XXX; /* group index or delta group */
    unsigned char affect;   /* specifies whether to affect mods, group, ptrbtn or controls*/
    unsigned char vmods1;    /* derived from vmods field of a modifier description */
    unsigned char vmods2;    /* derived from vmods field of a modifier description */
} XkbISOAction;
```

The *type* field of the `XkbISOAction` structure should always be `XkbSA_ISOLock`.

The interpretation of the *flags* field depends upon whether the `XkbSA_ISODfltsIsGroup` is set in the *flags* field or not.

If the `XkbSA_ISODfltsIsGroup` is set in the *flags* field, the action is used to change the group state. The remaining valid bits of the *flags* field are composed of a bitwise inclusive OR using the masks shown in Table 16.10.

Table 16.10 ISO Action Flags when XkbSA_ISODfltsIsGroup is Set

Flag	Meaning
<code>XkbSA_ISODfltsIsGroup</code>	If set, the action is used to change the base group state. Must be set for the remaining bits in this table to carry their interpretations.
<code>XkbSA_GroupAbsolute</code>	A key press sets the base group as specified by the <i>group_XXX</i> field and the <code>XkbSA_GroupAbsolute</code> bit of the <i>flags</i> field (see section Note). If no other actions are transformed by the <code>XkbISO_Lock</code> action, a key release locks the group. Otherwise, a key release clears group set by the key press.
<code>XkbSA_ISONoAffectMods</code>	If set, the <i>group_XXX</i> field represents an absolute group number. Otherwise, it represents a group delta to be added to the current group to determine the new group number.
<code>XkbSA_ISONoAffectGroup</code>	If set, any <code>XkbSA_SetGroup</code> or <code>XkbSA_LatchGroup</code> actions that occur simultaneously with the <code>XkbSA_ISOLock</code> action are treated as <code>XkbSA_LockGroup</code> actions instead.
<code>XkbSA_ISONoAffectPtr</code>	If set, any <code>XkbSA_PtrBtn</code> actions that occur simultaneously with the <code>XkbSA_ISOLock</code> action are treated as <code>XkbSA_LockPtrBtn</code> actions instead.
<code>XkbSA_ISONoAffectCtrls</code>	If set, any <code>XkbSA_SetControls</code> actions that occur simultaneously with the <code>XkbSA_ISOLock</code> action are treated as <code>XkbSA_LockControls</code> actions instead.

If the `XkbSA_ISODfltIsGroup` is not set in the `flags` field, the action is used to change the modifier state and the remaining valid bits of the `flags` field are composed of a bitwise inclusive OR using the masks shown in Table 16.11.

Table 16.11 ISO Action Flags when `XkbSA_ISODfltIsGroup` is Not Set

<i>Flag</i>	<i>Meaning</i>
<code>XkbSA_ISODfltIsGroup</code>	If not set, action is used to change the base modifier state. Must not be set for the remaining bits in this table to carry their interpretations.
	A key press sets the action modifiers in the keyboard's base modifiers using the <code>mask</code> , <code>real_mods</code> , <code>vmods1</code> , and <code>vmods2</code> fields (see section 16.1.3). If no other actions are transformed by the <code>XkbISO_Lock</code> action, a key release locks the action modifiers. Otherwise, a key release clears the base modifiers set by the key press.
<code>XkbSA_UseModMapMods</code>	If set, the action modifiers are determined by the modifiers bound by the modifier mapping of the key. Otherwise, the action modifiers are set to the modifiers specified by the <code>mask</code> , <code>real_mods</code> , <code>vmod1</code> , and <code>vmod2</code> fields.
<code>XkbSA_LockNoLock</code>	If set, the server only unlocks the action modifiers.
<code>XkbSA_LockNoUnlock</code>	If set, the server only locks the action modifiers.
<code>XkbSA_ISONoAffectMods</code>	If not set, any <code>XkbSA_SetMods</code> or <code>XkbSA_LatchMods</code> actions that occur simultaneously with the <code>XkbSA_ISOLock</code> action are treated as <code>XkbSA_LockMod</code> actions instead.
<code>XkbSA_ISONoAffectGroup</code>	If not set, any <code>XkbSA_SetGroup</code> or <code>XkbSA_LatchGroup</code> actions that occur simultaneously with the <code>XkbSA_ISOLock</code> action are treated as <code>XkbSA_LockGroup</code> actions instead.
<code>XkbSA_ISONoAffectPtr</code>	If not set, any <code>XkbSA_PtrBtn</code> actions that occur simultaneously with the <code>XkbSA_ISOLock</code> action are treated as <code>XkbSA_LockPtrBtn</code> actions instead.
<code>XkbSA_ISONoAffectCtrls</code>	If not set, any <code>XkbSA_SetControls</code> actions that occur simultaneously with the <code>XkbSA_ISOLock</code> action are treated as <code>XkbSA_LockControls</code> actions instead.

The `group_XXX` field represents a signed character. Xkb provides macros to convert between a signed integer value and a signed character as shown in section Note.

The `mask`, `real_mods`, `vmods1`, and `vmods2` fields represent the components of an Xkb modifier description (see section 7.2). While the `mask` and `real_mods` fields correspond directly to the `mask` and `real_mods` fields of an Xkb modifier description, the `vmods1` and `vmods2` fields are combined to correspond to the `vmods` field of an Xkb modifier description. Xkb provides macros to convert between the two formats as shown in section 16.1.3.

The `affect` field is composed of a bitwise inclusive OR using the masks shown in Table 16.11.

Table 16.12 ISO Action affect Field Values

<i>Affect</i>	<i>Meaning</i>
<code>XkbSA_ISODNoAffectMods</code>	If <code>XkbSA_ISONoAffectMods</code> is not set, any <code>SA_SetMods</code> or <code>SA_LatchMods</code> actions occurring simultaneously with the <code>XkbISOAction</code> are treated as <code>SA_LockMods</code> instead.

Table 16.12 ISO Action affect Field Values

<i>Affect</i>	<i>Meaning</i>
XkbSA_ISONoAffectGroup	If XkbSA_ISONoAffectGroup is not set, any SA_SetGroup or SA_LatchGroup actions occurring simultaneously with the XkbISOAction are treated as SA_LockGroup instead.
XkbSA_ISONoAffectPtr	If XkbSA_ISONoAffectPtr is not set, any SA_PtrBtn actions occurring simultaneously with the XkbISOAction are treated as SA_LockPtrBtn instead.
XkbSA_ISONoAffectCtrls	If XkbSA_ISONoAffectCtrls is not set, any SA_SetControls actions occurring simultaneously with the XkbISOAction are treated as SA_LockControls instead.

16.1.9 Actions for Changing the Active Screen

Actions associated with the XkbSwitchScreen action structure change the active screen on a multi-screen display:

Note 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 XkbSA_NoAction. Otherwise, key press and key release events do not generate an event.

```
typedef struct _XkbSwitchScreenAction {
    unsigned char      type;          /* XkbSA_SwitchScreen */
    unsigned char      flags;         /* controls screen switching */
    char              screenXXX;     /* screen number or delta */
} XkbSwitchScreenAction;
```

The *type* field of the XkbSwitchScreenAction structure should always be XkbSA_SwitchScreen.

The *flags* field is composed of the bitwise inclusive OR of the masks shown in Table 16.13.

Table 16.13 Switch Screen Action Flags

<i>Flag</i>	<i>Meaning</i>
XkbSA_SwitchAbsolute	If set, the <i>screenXXX</i> field represents the index of the new screen. Otherwise, it represents an offset from the current screen to the new screen.
XkbSA_SwitchApplication	If not set, the action should switch to another screen on the same server. Otherwise, it should switch to another X server or application that shares the same physical display.

The *screenXXX* field is a signed character value that represents either the relative or absolute screen index, depending upon the state of the XkbSA_SwitchAbsolute bit in the *flags* field. Xkb provides the following macros, to convert between the integer and signed character value for screen numbers in XkbSwitchScreenAction structures:

```
int XkbSAScreen(act)                  /* macro */
    XkbSwitchScreenAction act;        /* action from which to extract screen */
```

XkbSAScreen returns the *screenXXX* field of *act* converted to a signed int.

```

void XkbSASetScreen(act, s)          /* macro */
    XkbSwitchScreenAction act;        /* action in which to set screenXXX */
    int s;                          /* value to set in screenXXX */

```

XkbSASetScreen sets the *screenXXX* field of *act* from *s*.

16.1.10 Actions for Changing Boolean Controls State

Actions associated with the *XkbCtrlsAction* structure change the state of the boolean controls (see section 10.1):

```

typedef struct _XkbCtrlsAction {
    unsigned char type;           /* XkbSA_SetControls, XkbSA_LockControls */
    unsigned char flags;          /* with type, controls enabling and disabling of controls */
    unsigned char ctrls3;         /* ctrls0 through ctrls3 represent the boolean controls */
    unsigned char ctrls2;         /* ctrls0 through ctrls3 represent the boolean controls */
    unsigned char ctrls1;         /* ctrls0 through ctrls3 represent the boolean controls */
    unsigned char ctrls0;         /* ctrls0 through ctrls3 represent the boolean controls */
} XkbCtrlsAction;

```

The *type* field can have any one of the values shown in Table 16.14.

Table 16.14 Controls Action Types

Type	Effect
XkbSA_SetControls	<ul style="list-style-type: none"> A key press enables any boolean controls specified in the <i>ctrls</i> fields that were not already enabled at the time of the key press. A key release disables any controls enabled by the key press. This action can cause <i>XkbControlsNotify</i> events (see section 10.1).
XkbSA_LockControls	<ul style="list-style-type: none"> If the <i>XkbSA_LockNoLock</i> bit is not set in the <i>flags</i> field, a key press enables any controls specified in the <i>ctrls</i> fields that were not already enabled at the time of the key press. If the <i>XkbSA_LockNoUnlock</i> bit is not set in the <i>flags</i> field, a key release disables any controls specified in the <i>ctrls</i> fields that were not already disabled at the time of the key press. This action can cause <i>XkbControlsNotify</i> events (see section 10.1).

The *flags* field is composed of the bitwise inclusive OR of the masks shown in Table 16.15.

Table 16.15 Control Action Flags

Flag	Meaning
XkbSA_LockNoLock	If set, and the action type is <i>XkbSA_LockControls</i> , the server only disables controls.
XkbSA_LockNoUnlock	If set, and the action type is <i>XkbSA_LockControls</i> , the server only enables controls.

The *XkbSA_SetControls* action implements a key which enables a boolean control when pressed, and disables it when released. The *XkbSA_LockControls* action is used to implement a key which toggles the state of a boolean control each time it is pressed and released. The *XkbSA_LockNoLock* and *XkbSA_LockNoUnlock* flags allow modifying the toggling behavior to only unlock or only lock the boolean control.

The *ctrls0*, *ctrls1*, *ctrls2*, and *ctrls3* fields represent the boolean controls in the *enabled_ctrls* field of the controls structure (see section 10.1). Xkb provides the following macros, to convert between the two formats:

```
unsigned int XkbActionCtrls(act)          /* macro */
XkbCtrlsAction    act;                  /* action from which to extract controls */
```

XkbActionCtrls returns the *ctrls* fields of *act* converted to an unsigned int.

```
void XkbSAActionSetCtrls(act, ctrls)      /* macro */
XkbCtrlsAction    act;                  /* action in which to set ctrls0-ctrls3 */
unsigned int      ctrls;                /* value to set in ctrls0-ctrls3 */
```

XkbSAActionSetCtrls sets the *ctrls0* through *ctrls3* fields of *act* from *ctrls*.

16.1.11 Actions for Generating Messages

Actions associated with the *XkbMessageAction* structure generate *XkbActionMessage* events:

```
#define XkbActionMessageLength 6

typedef struct _XkbMessageAction {
    unsigned char type;        /* XkbSA_ActionMessage */
    unsigned char flags;       /* controls event generation via key presses and releases */
    unsigned char message[XkbActionMessageLength]; /* message */
} XkbMessageAction;
```

The *type* field of the *XkbMessageAction* structure should always be *XkbSA_ActionMessage*.

The *flags* field is composed of the bitwise inclusive OR of the masks shown in Table 16.16.

Table 16.16 Message Action Flags

Flag	Meaning
XkbSA_MessageOnPress	If set, key press events generate an <i>XkbActionMessage</i> event which reports the keycode, event type, and contents of the <i>message</i> field.
XkbSA_MessageOnRelease	If set, key release events generate an <i>XkbActionMessage</i> event which reports the keycode, event type, and contents of the <i>message</i> field.
XkbSA_MessageGenKeyEvent	If set, key press and key release events generate KeyPress and KeyRelease events, regardless of whether they generate <i>XkbActionMessage</i> events.

The *message* field is an array of *XkbActionMessageLength* unsigned characters, and may be set to anything the keymap designer wishes.

Detecting Key Action Messages

To receive *XkbActionMessage* events by calling either *XkbSelectEvents* or *XkbSelectEventDetails* (see section 4.3).

To receive *XkbActionMessage* events under all possible conditions, call *XkbSelectEvents* and pass *XkbActionMessageMask* in both *bits_to_change* and *values_for_bits*.

The `XkbActionMessage` event has no event details. However, you can call `XkbSelectEventDetails` using `XkbActionMessage` as the `event_type` and specifying `XkbAllActionMessageMask` in `bits_to_change` and `values_for_bits`. This has the same effect as a call to `XkbSelectEvents`.

The structure for the `XkbActionMessage` event is defined as follows:

```
typedef struct _XkbActionMessage {
    int             type;           /* Xkb extension base event code */
    unsigned long   serial;         /* X server serial number for event */
    Bool            send_event;     /* True => synthetically generated */
    Display *       display;        /* server connection where event generated */
    Time            time;          /* server time when event generated */
    int             xkb_type;       /* XkbActionMessage */
    int             device;         /* Xkb device id, will not be XkbUseCoreKbd */
    KeyCode         keycode;        /* keycode of key triggering event */
    Bool            press;          /* True => key press, False => release */
    Bool            key_event_follows; /* True => KeyPress/KeyRelease follows */
    char            message[XkbActionMessageLength+1]; /* message text */
} XkbActionMessageEvent;
```

The `keycode` is the keycode of the key that was pressed or released. The `press` field specifies whether the event was the result of a key press or key release.

The `key_event_follows` specifies whether a `KeyPress` (if `press` is `True`) or `KeyRelease` (if `press` is `False`) event is also sent to the client. As with all other Xkb events, `XkbActionMessageEvents` are delivered to all clients requesting them, regardless of the current keyboard focus. However, the `KeyPress` or `KeyRelease` event which conditionally follows an `XkbActionMessageEvent` is only sent to the client selected by the current keyboard focus. `key_event_follows` is `True` only for the client which is actually sent the following `KeyPress` or `KeyRelease` event.

The `message` field is set to the message specified in the action, and is guaranteed to be `NULL`-terminated; the Xkb extension forces a `NULL` into `message[XkbActionMessageLength]`.

16.1.12 Actions for Generating a Different Keycode

Actions associated with the `XkbRedirectKeyAction` structure generate `KeyPress` and `KeyRelease` events containing a keycode different from the key that was pressed or released:

```
typedef struct _XkbRedirectKeyAction {
    unsigned char   type;          /* XkbSA_RedirectKey */
    unsigned char   new_key;        /* keycode to be put in event */
    unsigned char   mods_mask;     /* mask of real mods to be reset */
    unsigned char   mods;          /* mask of real mods to take values from */
    unsigned char   vmods_mask0;   /* first half of mask of virtual mods to be reset */
    unsigned char   vmods_mask1;   /* other half of mask of virtual mods to be reset */
    unsigned char   vmods0;        /* first half of mask of virtual mods to take values from */
    unsigned char   vmods1;        /* other half of mask of virtual mods to take values from */
} XkbRedirectKeyAction;
```

The *type* field for the `XkbRedirectKeyAction` structure should always be `XkbSA_RedirectKey`.

Key presses cause a `KeyPress` event for the key specified by the *new_key* field instead of the actual key. The state reported in this event reports the current effective modifiers changed as follows: any real modifiers selected by the *mods_mask* field are set to corresponding values from the *mods* field. Any real modifiers bound to the virtual modifiers specified by the *vmods_mask0* and *vmods_mask1* fields are either set or cleared, depending upon the corresponding values in the *vmods0* and *vmods1* fields. If the real and virtual modifier definitions specify conflicting values for a single modifier, the real modifier definition has priority.

Key releases cause a `KeyRelease` event for the key specified by the *new_key* field instead of the actual key. The state for this event consists of the effective keyboard modifiers at the time of the release, changed as described above.

The `XkbSA_RedirectKey` action normally redirects to another key on the same device as the key that 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. (The input extension categorizes devices by breaking them into classes. Keyboards, and other input devices with keys, are classified as `KeyClass` devices by the input extension.)

The *vmods_mask0* and *vmods_mask1* fields actually represent one *vmods_mask* value, as described in Chapter 7. Xkb provides the following macros, to convert between the two formats:

```
unsigned int XkbSARedirectVModsMask(act)      /* macro */
XkbRedirectKeyAction    act;                      /* action from which to extract vmods */
```

XkbSARedirectVModsMask returns the *vmods_mask0* and *vmods_mask1* fields of *act* converted to an unsigned int.

```
void XkbSARedirectSetVModsMask(act, vm)      /* macro */
XkbRedirectKeyAction    act;                      /* action in which to set vmods */
unsigned int            vm;                      /* new value for virtual modifier mask */
```

XkbSARedirectSetVModsMask sets the *vmods_mask0* and *vmods_mask1* fields of *act* from *vm*.

Similarly, the *vmods0* and *vmods1* fields actually represent one *vmods* value, as described in Chapter 7. To convert between the two formats, Xkb provides the following convenience macros:

```
unsigned int XkbSARedirectVMods(act)           /* macro */
XkbRedirectKeyAction    act;                      /* action from which to extract vmods */
```

XkbSARedirectVMods returns the *vmods0* and *vmods1* fields of *act* converted to an unsigned int.

```
void XkbSARedirectSetVMods(act, vm)           /* macro */
XkbRedirectKeyAction    act;                      /* action in which to set vmods */
unsigned int            v;                       /* new value for virtual modifiers */
```

XkbSARedirectSetVMods sets the *vmods0* and *vmods1* of *act* from *v*.

16.1.13 Actions for Generating DeviceButtonPress and DeviceButtonRelease

Actions associated with `XkbDeviceBtnAction` structures generate `DeviceButtonPress` and `DeviceButtonRelease` events instead of normal `KeyPress` and `KeyRelease` events:

```
typedef struct _XkbDeviceBtnAction {
    unsigned char type;      /* XkbSA_DeviceBtn, XkbSA_LockDeviceBtn */
    unsigned char flags;     /* with type, specifies locking or unlocking */
    unsigned char count;     /* controls number of DeviceButtonPress and Release events */
    unsigned char button;    /* index of button on device */
    unsigned char device;    /* device id of an X input extension device */
} XkbDeviceBtnAction;
```

The `type` field can have any one of the values shown in Table 16.17.

Table 16.17 Device Button Action Types

Type	Effect
<code>XkbSA_DeviceBtn</code>	<ul style="list-style-type: none"> 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 <code>XkbSA_NoAction</code>. Otherwise, key presses cause one or more input extension device events instead of the usual key press event. If the <code>count</code> field is 0, a key press generates a single <code>DeviceButtonPress</code> event. If <code>count</code> is greater than 0, a key press event generates <code>count</code> pairs of <code>DeviceButtonPress</code> and <code>DeviceButtonRelease</code> events. If <code>count</code> is 0, a key release generates an input extension <code>DeviceButtonRelease</code> event that matches the event generated by the corresponding key press. If <code>count</code> is non-zero, a key release does not cause a <code>DeviceButtonRelease</code> event. Key releases never cause <code>KeyRelease</code> events.
<code>XkbSA_LockDeviceBtn</code>	<ul style="list-style-type: none"> If the device or button specified by this action are illegal, this action behaves like <code>XkbSA_NoAction</code>. Otherwise, if the specified button is not locked and the <code>XkbSA_LockNoLock</code> bit is not set in the <code>flags</code> field, a key press generates an input extension <code>DeviceButtonPress</code> event instead of a <code>KeyPress</code> event and locks the button. If the button is already locked or if <code>XkbSA_LockNoLock</code> bit is set in the <code>flags</code> field, the key press is ignored and has no effect. If the corresponding key press was ignored, and if the <code>XkbSA_LockNoUnlock</code> bit is not set in the <code>flags</code> field, a key release generates an input extension <code>DeviceButtonRelease</code> event instead of a <code>KeyRelease</code> event and unlocks the button. If the corresponding key press locked a button, the key release is ignored and has no effect.

The `flags` field is composed of the bitwise inclusive OR of the masks shown in Table 16.18.

Table 16.18 Device Button Action Flags

Flag	Meaning
<code>XkbSA_LockNoLock</code>	If set, and the action type is <code>XkbSA_LockDeviceBtn</code> , the server only unlocks the button.

Table 16.18 Device Button Action Flags

<i>Flag</i>	<i>Meaning</i>
XkbSA_LockNoUnlock	If set, and the action type is XkbSA_LockDeviceBtn, the server only locks the button.

16.1.14 Actions for Simulating Events from Device Valuators

A *valuator* manipulates a range of values for some entity, like a mouse axis, a slider or a dial. Actions associated with XkbDeviceValuatorAction structures are used to simulate events from one or two input extension device valuators.

```
typedef struct _XkbDeviceValuatorAction {
    unsigned char type;      /*XkbSA_DeviceValuator */
    unsigned char device;    /* device id */
    unsigned char v1_what;   /* determines how valuator is to behave for valuator 1 */
    unsigned char v1_ndx;    /* specifies a real valuator */
    unsigned char v1_value;  /* the value for valuator 1 */
    unsigned char v2_what;   /* determines how valuator is to behave for valuator 2 */
    unsigned char v2_ndx;    /* specifies a real valuator */
    unsigned char v2_value;  /* the value for valuator 1 */
} XkbDeviceValuatorAction;
```

If *device* is illegal or if neither *v1_ndx* nor *v2_ndx* specifies a legal valuator, this action behaves like XkbSA_NoAction.

The low four bits of *v1_what* and *v2_what* specify the corresponding scale value (denoted *val<n>Scale* in Table 16.17), if needed. The high four bits of *v1_what* and *v2_what* specify the operation to perform to set the values. The high four bits of *v1_what* and *v2_what* can have the values shown in Table 16.17; the use of *val<n>Scale* is shown in that table also.

Table 16.19 Device Valuator v<n>_what High Bits Values

<i>Value of high bits</i>	<i>Effect</i>
XkbSA_IgnoreVal	No action
XkbSA_SetValMin	<i>v<n>_value</i> is set to its minimum legal value.
XkbSA_SetValCenter	<i>v<n>_value</i> is centered (to (max-min)/2).
XkbSA_SetValMax	<i>v<n>_value</i> is set to its maximum legal value.
XkbSA_SetValRelative	<i>v<n>_value</i> * ($2^{\text{val<n>Scale}}$) is added to <i>v<n>_value</i> .
XkbSA_SetValAbsolute	<i>v<n>_value</i> is set to ($2^{\text{val<n>Scale}}$)

Illegal values for XkbSA_SetValRelative or XkbSA_SetValAbsolute are clamped into range. Note that all of these possibilities are legal for absolute valuators. For relative valuators, only XkbSA_SetValRelative is permitted. Part of the input extension description of a device is the range of legal values for all absolute valuators, whence the maximum and minimum legal values shown in Table 16.17.

The following two masks are provided as a convenience to select either portion of *v1_what* or *v2_what*:

```
#define XkbSA_ValOpMask      (0x70)
#define XkbSA_ValScaleMask    (0x07)
```

v1_ndx and *v2_ndx* specify valuators that actually exists. For example, most mice have two valuators (x and y axes) so the only legal values for a mouse would be 0 and 1. For a dial box with eight dials, any value in the range 0..7 would be correct.

16.1.15 Obtaining Key Actions for Keys from the Server

To update the actions (the *key_acts* array) for a subset of the keys in a keyboard description, call *XkbGetKeyActions*.

```
Status XkbGetKeyActions(dpy, first, num, xkb)
    Display *          dpy;      /* connection to X server */
    unsigned int       first;    /* keycode of first key of interest */
    unsigned int       num;     /* number of keys desired */
    XkbDescPtr        xkb;     /* pointer to keyboard description where result is stored */
```

XkbGetKeyActions sends a request to the server to obtain the actions for *num* keys on the keyboard starting with key *first*. It waits for a reply, and returns the actions in the *server->key_acts* field of *xkb*. If successful, *XkbGetKeyActions* returns Success. The *xkb* parameter must be a pointer to a valid Xkb keyboard description.

If the *server* map in the *xkb* parameter has not been allocated, *XkbGetKeyActions* allocates and initializes it before obtaining the actions.

If the server does not have a compatible version of Xkb, or the Xkb extension has not been properly initialized, *XkbGetKeyActions* returns BadAccess. If *num* is less than 1 or greater than *XkbMaxKeyCount*, *XkbGetKeyActions* returns BadValue. If any allocation errors occur, *XkbGetKeyActions* returns BadAlloc.

16.1.16 Changing the Number of Actions Bound to a Key

To change the number of actions bound to a key, use *XkbResizeKeyAction*.

```
XkbAction *XkbResizeKeyActions(xkb, key, needed)
    XkbDescRec *      xkb;      /* keyboard description to change */
    int              key;      /* keycode of key to change */
    int              needed;   /* new number of actions required */
```

The *xkb* parameter points to the keyboard description containing the *key* whose number of actions is to be changed. The *key* parameter is the keycode of the key to change, and *needed* specifies the new number of actions required for the key.

XkbResizeKeyActions reserves the space needed for the actions and returns a pointer to the beginning of the new array that holds the actions. It can change the *acts*, *num_acts*, and *size_acts* fields of *xkb->server* if it is necessary to reallocate the *acts* array.

If *needed* is greater than the current number of keysyms for the key, *XkbResizeKeyActions* initializes all new actions in the array to NoAction.

Since the number of actions needed by a key is normally computed as width * number of groups, and *XkbResizeKeyActions* does not modify either the width or number of groups for the key, a discrepancy exists upon return from *XkbResizeKeyActions* between the space allocated for the actions and the number required. The unused entries in the list of actions returned by *XkbResizeKeyActions* are not preserved across future calls to any of the map editing functions, so you must update the key actions (which updates the width

and number of groups for the key) before calling another allocator function. A call to *XkbChangeTypesOfKey* updates these.

If any allocation errors occur while resizing the number of actions bound to the key, *XkbResizeKeyActions* returns NULL.

Note A change to the number of actions bound to a key should be accompanied by a change in the number of symbols bound to a key. Refer to section 15.3.7 on page 138 for more information on changing the number of symbols bound to a key.

16.2 Key Behavior

Key behavior refers to the demeanor of a key. For example, the expected behavior of the CapsLock key is that it logically locks when pressed, and then logically unlocks when pressed again.

16.2.1 Radio Groups

Keys that belong to the same radio group have the *XkbKB_RadioGroup* type in the *type* field and the radio group index specified in the *data* field in the *XkbBehavior* structure. If the radio group has a name in the *XkbNamesRec* structure, the radio group index is the index into the *radio_group* array in the *XkbNamesRec* structure. A radio group key when pressed stays logically down until another key in the radio group is pressed, when the first key becomes logically up and the new key becomes logically down. Setting the *XkbKB_RGAllowNone* bit in the behavior for all of the keys of the radio group means that pressing the logically-down member of the radio group causes it to logically release, in which case none of the keys of the radio group would be logically down. If *XkbKB_RGAllowNone* is not set, there is no way to release the logically-down member of the group.

The low five bits of the *data* field of the *XkbBehavior* structure are the group number, the high three bits are flags. The only flag currently defined is:

```
#define XkbRG_AllowNone 0x80
```

16.2.2 The XkbBehavior Structure

The *behaviors* field of the server map is an array of *XkbBehavior* structures, indexed by keycode, and contains the behavior for each key. The *XkbBehavior* structure is defined as follows:

```
typedef struct _XkbBehavior {
    unsigned char type;      /* behavior type + optional XkbKB_Permanent bit */
    unsigned char data;
} XkbBehavior;
```

The *type* field specifies the Xkb behavior, and the value of the *data* field depends upon the *type*. Xkb supports the key behaviors shown in Table 16.20.

Table 16.20 Key Behaviors

Type	Effect
XkbKB_Default	Press and release events are processed normally. The <i>data</i> field is unused.

Table 16.20 Key Behaviors

Type	Effect
XkbKB_Lock	If a key is logically up (that is, 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. The <i>data</i> field is unused.
XkbKB_RadioGroup	If another member of the radio group is logically down (all members of the radio group have the same index, specified in <i>data</i>) 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 Xkb_RGA1lowNone bit in <i>flags</i> . If it is set, the key release is processed normally; otherwise the key release is also ignored. All other key release events are ignored.
XkbKB_Overlay1	If the Overlay1 control is enabled (see section 10.4), <i>data</i> is interpreted as a keycode, and events from this key are reported as if they came from <i>data</i> 's keycode. Otherwise, press and release events are processed normally.
XkbKB_Overlay2	If the Overlay2 control is enabled (see section 10.4), <i>data</i> is interpreted as a keycode, and events from this key are reported as if they came from <i>data</i> 's keycode. Otherwise, press and release events are processed normally.

Xkb also provides the mask, XkbKB_Permanent, to specify whether the key behavior type should be simulated by Xkb, or if the key behavior describes an unalterable physical, electrical, or software aspect of the keyboard. If the XkbKB_Permanent bit is not set in the *type* field, Xkb simulates the behavior in software. Otherwise, Xkb relies upon the keyboard to implement the behavior.

16.2.3 Obtaining Key Behaviors for Keys from the Server

To obtain the behaviors (the *behaviors* array) for a subset of the keys in a keyboard description from the server, use *XkbGetKeyBehaviors*:

Status **XkbGetKeyBehaviors**(*dpy*, *first*, *num*, *xkb*)

Display *	<i>dpy</i> ;	/* connection to server */
unsigned int	<i>first</i> ;	/* keycode of first key to get */
unsigned int	<i>num</i> ;	/* number of keys for which behaviors are desired */
XkbDescPtr	<i>xkb</i> ;	/* Xkb description to contain the result */

XkbGetKeyBehaviors sends a request to the server to obtain the behaviors for *num* keys on the keyboard starting with the key whose keycode is *first*. It waits for a reply, and returns the behaviors in the *server->behaviors* field of *xkb*. If successful, *XkbGetKeyBehaviors* returns Success.

If the *server* map in the *xkb* parameter has not been allocated, *XkbGetKeyBehaviors* allocates and initializes it before obtaining the actions.

If the server does not have a compatible version of Xkb, or the Xkb extension has not been properly initialized, *XkbGetKeyBehaviors* returns BadAccess. If *num* is less than 1 or greater than XkbMaxKeyCount, *XkbGetKeyBehaviors* returns BadValue. If any allocation errors occur, *XkbGetKeyBehaviors* returns BadAlloc.

16.3 Explicit Components—Avoiding Automatic Remapping by the Server

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 (See section 17.1.2 for more information on how Xkb chooses the default values).

This automatic remapping might replace definitions explicitly requested by an application, so the Xkb keyboard description defines an explicit components mask for each key. Any aspects of the automatic remapping which are listed in the explicit components mask for a key are not changed by the automatic keyboard mapping.

The explicit components masks are held in the *explicit* field of the server map, which is an array indexed by keycode. Each entry in this array is a mask that is a bitwise inclusive OR of the values shown in Table 16.21.

Table 16.21 Explicit Component Masks

<i>Bit in Explicit Mask</i>	<i>Value</i>	<i>Protects Against</i>
ExplicitKeyType1	(1<<0)	Automatic determination of the key type associated with Group1.
ExplicitKeyType2	(1<<1)	Automatic determination of the key type associated with Group2.
ExplicitKeyType3	(1<<2)	Automatic determination of the key type associated with Group3.
ExplicitKeyType4	(1<<3)	Automatic determination of the key type associated with Group4.
ExplicitInterpret	(1<<4)	Application of any of the fields of a symbol interpretation to the key in question.
ExplicitAutoRepeat	(1<<5)	Automatic determination of auto-repeat status for the key, as specified in a symbol interpretation.
ExplicitBehavior	(1<<6)	Automatic assignment of the XkbKB_Lock behavior to the key, if the XkbSI_LockingKey flag is set in a symbol interpretation.
ExplicitVModMap	(1<<7)	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.

16.3.1 Obtaining Explicit Components for Keys from the Server

To obtain the explicit components (the *explicit* array) for a subset of the keys in a keyboard description, use *XkbGetKeyExplicitComponents*.

```
Status XkbGetKeyExplicitComponents(dpy, first, num, xkb)
Display *           dpy;          /* connection to server */
unsigned int        first;        /* keycode of first key to fetch */
unsigned int        num;         /* number of keys for which to get explicit info */
XkbDescPtr         xkb;         /* Xkb description in which to put results */
```

XkbGetKeyExplicitComponents sends a request to the server to obtain the explicit components for *num* keys on the keyboard starting with key *first*. It waits for a reply, and returns the explicit components in the *server->explicit* array of *xkb*. If successful, *XkbGetKeyExplicitComponents* returns Success. The *xkb* parameter must be a pointer to a valid Xkb keyboard description.

If the *server* map in the *xkb* parameter has not been allocated, *XkbGetKeyExplicitComponents* allocates and initializes it before obtaining the actions.

If the server does not have a compatible version of Xkb, or the Xkb extension has not been properly initialized, *XkbGetKeyExplicitComponents* returns *BadMatch*. If *num* is less than 1 or greater than *XkbMaxKeyCount*, *XkbGetKeyExplicitComponents* returns *BadValue*. If any allocation errors occur, *XkbGetKeyExplicitComponents* returns *BadAlloc*.

16.4 Virtual Modifier Mapping

The *vmods* member of the server map is a fixed-length array containing *XkbNumVirtualMods* entries. Each entry corresponds to a virtual modifier and provides the binding of the virtual modifier to the real modifier bits. Each entry in the *vmods* array is a bitwise inclusive OR of the legal modifier masks:

```
ShiftMask  
LockMask  
ControlMask  
Mod1Mask  
Mod2Mask  
Mod3Mask  
Mod4Mask  
Mod5Mask
```

The *vmodmap* member of the server map is similar to the *modmap* array of the client map (see section 15.4), but is used to define the virtual modifier mapping for each key. Like the *modmap* member, it is indexed by keycode, and each entry is a mask representing the virtual modifiers bound to the corresponding key:

- Each of the bits in a *vmodmap* entry represents an index into the *vmods* member. That is, bit 0 of a *vmodmap* entry refers to index 0 of the *vmods* array, bit 1 refers to index 1, and so on.
- If a bit is set in the *vmodmap* entry for a key, that key is bound to the corresponding virtual modifier in the *vmods* array.

The *vmodmap* and *vmods* members of the server map are the “master” virtual modifier definitions. Xkb automatically propagates any changes to these fields to all other fields that use virtual modifier mappings.

The overall relationship of fields dealing with virtual modifiers in an Xkb keyboard description are shown in Figure 16.2.

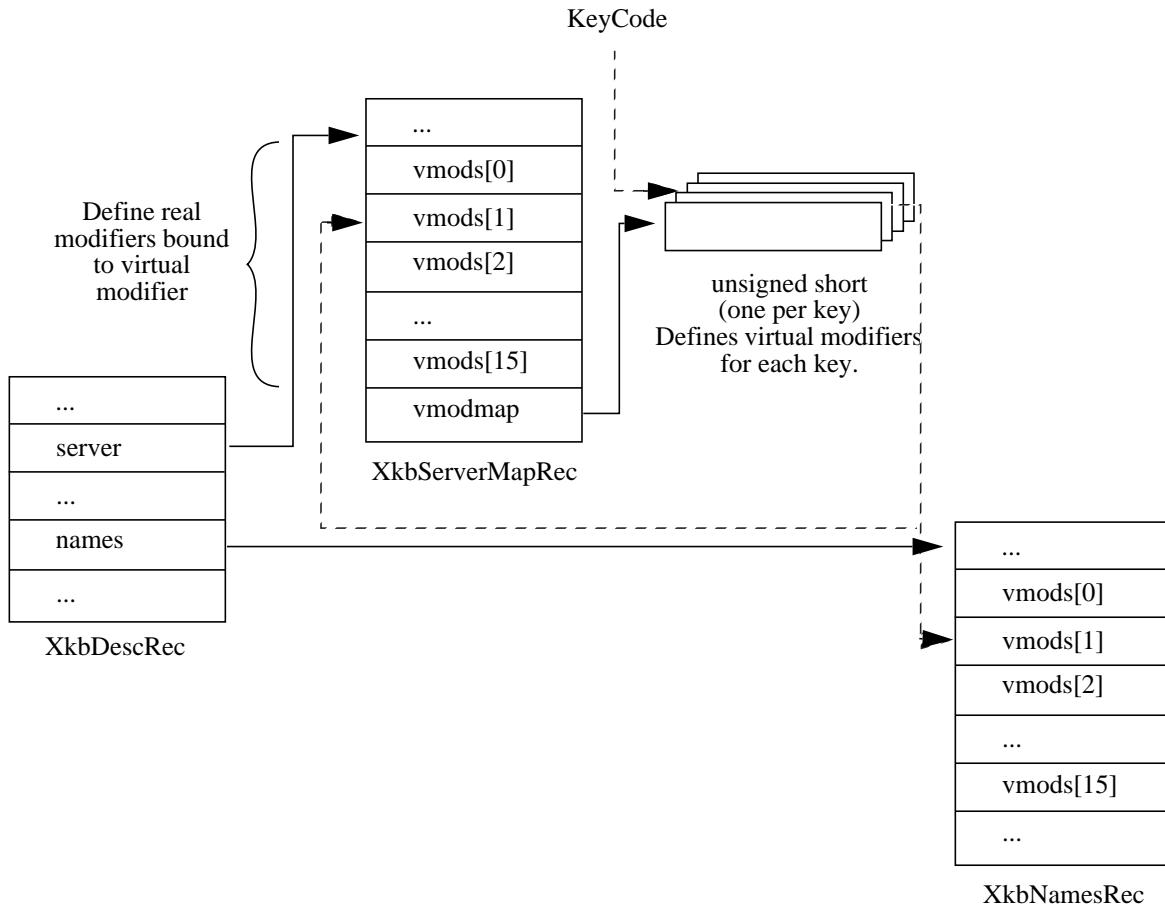


Figure 16.2 Virtual Modifier Relationships

16.4.1 Obtaining Virtual Modifier Bindings from the Server

To obtain a subset of the virtual modifier bindings (the *vmods* array) in a keyboard description, use *XkbGetVirtualMods*:

```
Status XkbGetVirtualMods(dpy, which, xkb)
    Display *          dpy;           /* connection to server */
    unsigned int        which;         /* mask indicating virtual modifier bindings to get */
    XkbDescPtr         xkb;          /* Xkb description where results will be placed */
```

XkbGetVirtualMods sends a request to the server to obtain the *vmods* entries for the virtual modifiers specified in the mask, *which*, and waits for a reply. See section 7.1 on page 31 for a description of how to determine the virtual modifier mask. For each bit set in *which*, *XkbGetVirtualMods* updates the corresponding virtual modifier definition in the *server->vmods* array of *xkb*. The *xkb* parameter must be a pointer to a valid Xkb keyboard description. If successful, *XkbGetVirtualMods* returns Success.

If the *server* map has not been allocated in the *xkb* parameter, *XkbGetVirtualMods* allocates and initializes it before obtaining the virtual modifier bindings.

If the server does not have a compatible version of Xkb, or the Xkb extension has not been properly initialized, *XkbGetVirtualMods* returns *BadMatch*. Any errors in allocation cause *XkbGetVirtualMods* to return *BadAlloc*.

16.4.2 Obtaining Per-Key Virtual Modifier Mappings from the Server

To obtain the virtual modifier map (the *vmodmap* array) for a subset of the keys in a keyboard description, use *XkbGetKeyVirtualModMap*:

Status **XkbGetKeyVirtualModMap**(*dpy*, *first*, *num*, *xkb*)

Display *	<i>dpy</i> ;	/* connection to server */
unsigned int	<i>first</i> ;	/* keycode of first key to fetch */
unsigned int	<i>num</i> ;	/* # keys for which virtual mod maps are desired */
XkbDescPtr	<i>xkb</i> ;	/* Xkb description where results will be placed */

XkbGetKeyVirtualModMap sends a request to the server to obtain the virtual modifier mappings for *num* keys on the keyboard starting with key *first*. It waits for a reply, and returns the virtual modifier mappings in the *server->vmodmap* array of *xkb*. If successful, *XkbGetKeyVirtualModMap* returns *Success*. The *xkb* parameter must be a pointer to a valid Xkb keyboard description.

If the *server* map in the *xkb* parameter has not been allocated, *XkbGetKeyVirtualModMap* allocates and initializes it before obtaining the virtual modifier mappings.

If the server does not have a compatible version of Xkb, or the Xkb extension has not been properly initialized, *XkbGetKeyVirtualModMap* returns *BadMatch*. If *num* is less than 1 or greater than *XkbMaxKeyCode*, *XkbGetKeyVirtualModMap* returns *BadValue*. If any allocation errors occur, *XkbGetKeyVirtualModMap* returns *BadAlloc*.

17 The Xkb Compatibility Map

As shown in the diagram below, the X server is normally dealing with more than one client, each of which may be receiving events from the keyboard, and each of which may issue requests to modify the keyboard in some manner. Each client may be either Xkb-unaware, Xkb-capable, or Xkb-aware. The server itself may be either Xkb-aware or Xkb-unaware. If the server is Xkb-unaware, Xkb state and keyboard mappings are not involved in any manner, and Xkb-aware clients may not issue Xkb requests to the server. If the server is Xkb-aware, the server must be able to deliver events and accept requests in which the keyboard state and mapping are compatible with the mode in which the client is operating. Consequently, for some situations, conversions must be made between Xkb state / keyboard mappings and core protocol state / keyboard mappings, and vice-versa.

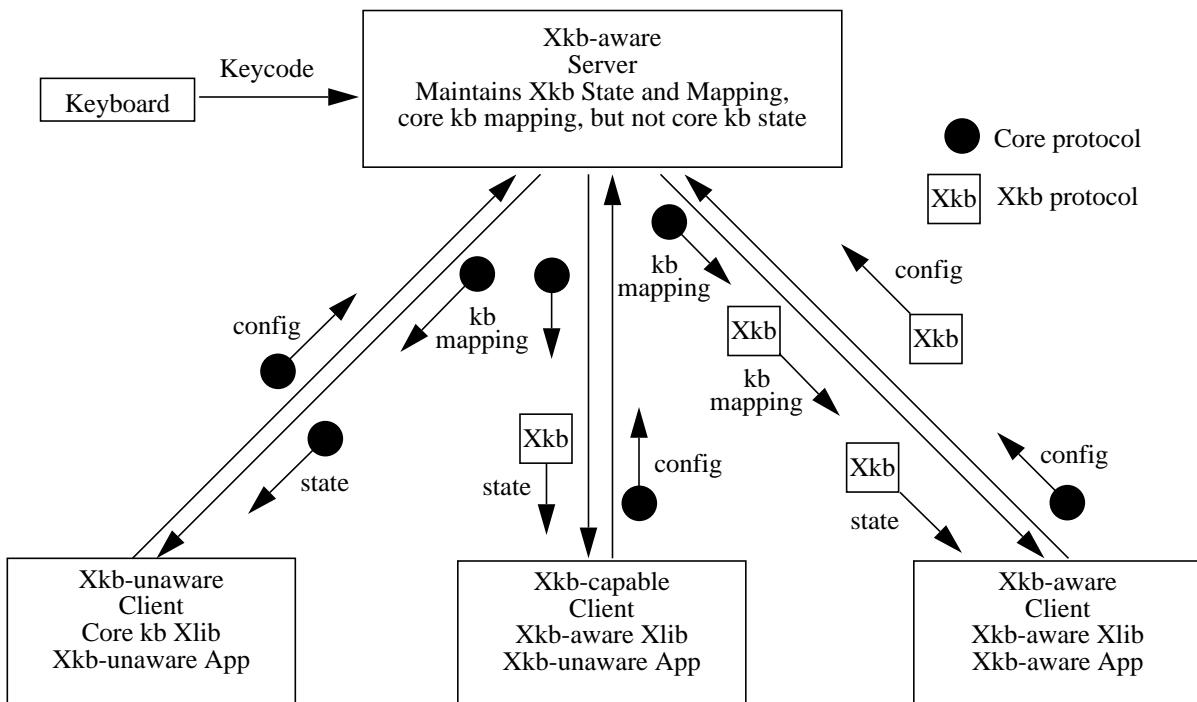


Figure 17.1 Server Interaction with Types of Clients

In addition to these situations involving a single server, there are cases where a client which deals with multiple servers may need to configure keyboards on different servers to be similar and the different servers may not all be Xkb-aware. Finally, a client may be dealing with descriptions of keyboards (files, etc.) which are based on core protocol, and therefore may need to be able to map these descriptions to Xkb descriptions.

An Xkb-aware server maintains keyboard state and mapping as an Xkb keyboard state and an Xkb keyboard mapping, plus a compatibility map used to convert from Xkb components to core components and vice-versa. In addition, the server also maintains a core keyboard mapping which approximates the Xkb keyboard mapping. The core keyboard mapping may be updated piecemeal, on a per-key basis. When the server receives a core protocol `ChangeKeyboardMapping` or `SetModifierMapping` request, it updates its core keyboard mapping, then uses the compatibility map to update its Xkb keyboard map-

ping. When the server receives an `XkbSetMap` request, it updates those portions of its Xkb keyboard mapping specified by the request, then uses its compatibility map to update the corresponding parts of its core keyboard map. Consequently, the server's Xkb keyboard map and also its core keyboard map may contain components which were set directly, and others which were computed. Figure 17.2 illustrates these relationships.

Note The core keyboard map is contained only in the server, not in any client-side data structures.

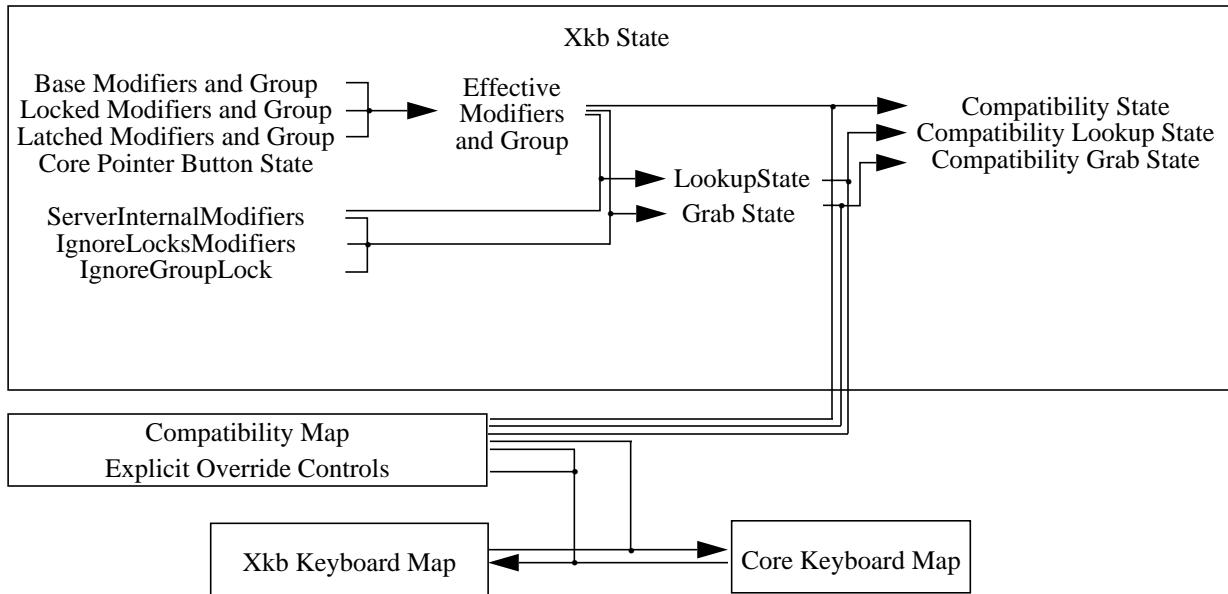


Figure 17.2 Server Derivation of State and Keyboard Mapping Components

There are three kinds of compatibility transformations made by the server:

1. Xkb State to Core State

Keyboard state information reported to a client in the state field of various core events may be translated from the Xkb keyboard state maintained by the server, which includes a group number, to core protocol state, which does not.

In addition, whenever the Xkb state is retrieved, the `compat_state`, `compat_grab_mods`, and `compat_lookup_mods` fields of the `XkbStateRec` returned indicate the result of applying the compatibility map to the current Xkb state in the server.

2. Core Keyboard Mapping to Xkb Keyboard Mapping

After core protocol requests received by the server to change the keyboard mapping (`ChangeKeyboardMapping` and `SetModifierMapping`) have been applied to the server's core keyboard map, the results must be transformed to achieve an equivalent change of the Xkb keyboard mapping maintained by the server.

3. Xkb Keyboard Mapping to Core Keyboard Mapping

After Xkb protocol requests received by the server to change the keyboard mapping (`XkbSetMap`) have been applied to the server's Xkb keyboard map, the results are

transformed to achieve an approximately equivalent change to the core keyboard mapping maintained by the server.

This chapter discusses how a client may modify the compatibility map so that subsequent transformations have a particular result.

17.1 The XkbCompatMap Structure

All configurable aspects of mapping Xkb state and configuration to and from core protocol state and configuration are defined by a compatibility map, contained in an `XkbCompatMap` structure; plus a set of explicit override controls used to prevent particular components of type 2 (core-to-Xkb keyboard mapping) transformations from automatically occurring. These explicit override controls are maintained in a separate data structure discussed in section 16.3.

The *compat* member of an Xkb keyboard description (`XkbDescRec`) points to the `XkbCompatMap` structure:

```
typedef struct _XkbCompatMapRec {
    XkbSymInterpretPtr    sym_interpret;      /* symbol based key semantics*/
    XkbModsRec            groups[XkbNumKbdGroups];    /* group => modifier map */
    unsigned short          num_si;           /* # structures used in sym_interpret */
    unsigned short          size_si;           /* # structures allocated in sym_interpret */
} XkbCompatMapRec, *XkbCompatMapPtr;
```

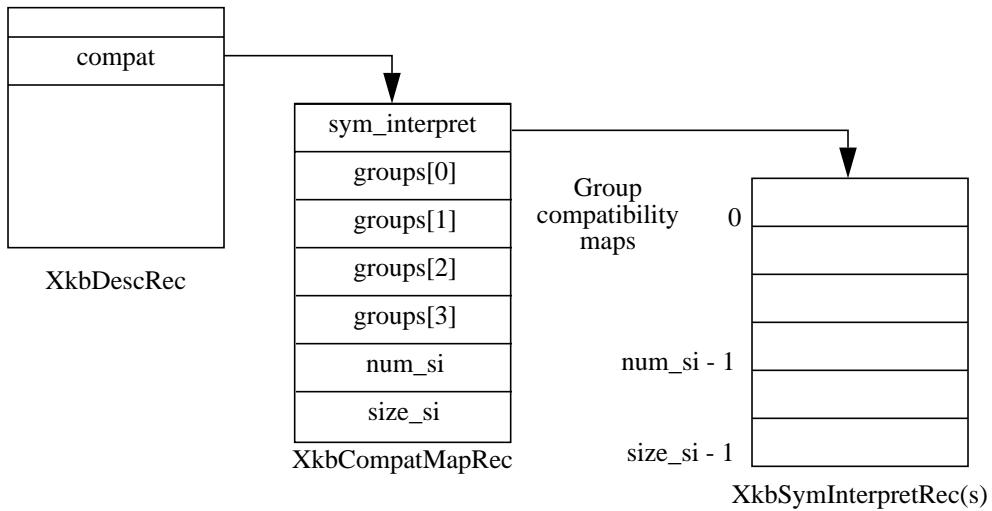


Figure 17.3 Xkb Compatibility Data Structures

The subsections below discuss how the compatibility map and explicit override controls are used in each of the three cases where compatibility transformations are made.

17.1.1 Xkb State to Core Protocol State Transformation

As shown in Figure 17.3, there are four *group compatibility maps* (contained in `groups [0..3]`) in the `XkbCompatMapRec` structure, one per possible Xkb group. Each group compatibility map is a modifier definition (see section 7.2 for a description of modifier defini-

tions). The *mask* component of the definition specifies which real modifiers should be set in the core protocol state field when the corresponding group is active. Since only one group is active at any one time, only one of the four possible transformations is ever applied at any one point in time. If the device described by the `XkbDescRec` does not support four groups, the extra `groups` fields are present, but undefined.

Normally, the Xkb-aware server reports keyboard state in the *state* member of events such as a `KeyPress` event and `ButtonPress` event, encoded as follows:

bits	meaning
15	0
13-14	Group index
8-12	Pointer Buttons
0-7	Modifiers

For Xkb-unaware clients, only core protocol keyboard information may be reported. Since core protocol does not define the group index, the group index is mapped to modifier bits as specified by the `groups[group index]` field of the compatibility map (the bits set in the compatibility map are ORed into bits 0-7 of the state), and bits 13-14 are reported in the event as zero.

17.1.2 Core Keyboard Mapping to Xkb Keyboard Mapping Transformation

When a core protocol keyboard mapping request is received by the server, the server's core keyboard map is updated, and then the Xkb map maintained by the server is updated. Since a client may have explicitly configured some of the Xkb keyboard mapping in the server, this automatic regeneration of the Xkb keyboard mapping from the core protocol keyboard mapping should not modify any components of the Xkb keyboard mapping which were explicitly set by a client. The client must set explicit override controls to prevent this from happening (see section 16.3). The core-to-Xkb mapping is done as follows:

1. Map the symbols from the keys in the core keyboard map to groups and symbols on keys in the Xkb keyboard map. The core keyboard mapping is of fixed width, so each key in the core mapping has the same number of symbols associated with it. The Xkb mapping allows a different number of symbols to be associated with each key; those symbols may be divided into a different number of groups (1-4) for each key. For each key, this process therefore involves partitioning the fixed number of symbols from the core mapping into a set of variable length groups with a variable number of symbols in each group. For example, if the core protocol map is of width five, the partition for one key might result in one group with two symbols and another with three symbols. A different key might result in two groups with two symbols plus a third group with one symbol. The core protocol map requires at least two symbols in each of the first two groups.
 - 1a. For each changed key, determine the number of groups represented in the new core keyboard map. This results in a tentative group count for each key in the Xkb map.
 - 1b. For each changed key, determine the number of symbols in each of the groups found in step 1a.. There is one explicit override control associated with each of the four possible groups for each Xkb key, `ExplicitKeyType1` through `ExplicitKeyType4`. If no explicit override control is set for a group, the number of symbols used for that group from the core map is two. If the explicit override control is set for a group on the key, the number of symbols used for that Xkb

group from the core map is the width of the Xkb group with one exception: because of the core protocol requirement for at least two symbols in each of groups one and two, the number of symbols used for groups one and two is the maximum of 2 or the width of the Xkb group.

- 1c. For each changed key, assign the symbols in the core map to the appropriate group on the key. If the total number of symbols required by the Xkb map for a particular key needs more symbols than the core protocol map contains, the additional symbols are taken to be `NoSymbol` keysyms appended to the end of the core set. If the core map contains more symbols than are needed by the Xkb map, trailing symbols in the core map are discarded. In the absence of an explicit override for group one or two, symbols are assigned in order by group; the first symbols in the core map are assigned to group one, in order, followed by group two, etc. For example, if the core map contained eight symbols per key, and a particular Xkb map contained 2 symbols for G1 and G2 and three for G3, the symbols would be assigned as (G is group, L is shift level):

G1L1 G1L2 G2L1 G2L2 G3L1 G3L2 G3L3

If an explicit override control is set for group one or two, the symbols are taken from the core set in a somewhat different order. The first four symbols from the core set are assigned to G1L1, G1L2, G2L1, G2L2, respectively. If group one requires more symbols, they are taken next, and then any additional symbols needed by group two. Group three and four symbols are taken in complete sequence after group two. For example, a key with four groups and three symbols in each group would take symbols from the core set in the following order:

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

As noted above, the core protocol map requires at least two symbols in groups one and two. Because of this, if an explicit override control for an Xkb key is set and group one and / or group two is of width one, it is not possible to generate the symbols taken from the core protocol set and assigned to position G1L2 and / or G2L2.

- 1d. For each group on each changed key, assign a key type appropriate for the symbols in the group.
- 1e. For each changed key, remove any empty or redundant groups.

At this point, the groups and their associated symbols have been assigned to the corresponding key definitions in the Xkb map.

2. Apply symbol interpretations to modify key operation. This phase is completely skipped if the `ExplicitInterpret` override control bit is set in the explicit controls mask for the Xkb key (see section 16.3).
 - 2a. For each symbol on each changed key, attempt to match the symbol and modifiers from the Xkb map to a symbol interpretation describing how to generate the symbol.
 - 2b. When a match is found in step 2a., apply the symbol interpretation to change the semantics associated with the symbol in the Xkb key map. If no match is found, apply a default interpretation.

The symbol interpretations used in step 2 are configurable, and may be specified using `XkbSymInterpretRec` structures referenced by the `sym_interpret` field of an `XkbCompatMapRec` (see Figure 17.3).

Symbol Interpretations — the `XkbSymInterpretRec` Structure

Symbol interpretations are used to guide the X server when it modifies the Xkb keymap in step 2. above. An initial set of symbol interpretations is loaded by the server when it starts. A client may add new ones using `XkbSetCompatMap` (see section 17.4).

Symbol interpretations result in key semantics being set. When a symbol interpretation is applied, the following components of server key event processing may be modified for the particular key involved:

- Virtual modifier map
- Auto repeat
- Key behavior (may be set to `XkbKB_Lock`)
- Key action (see section 16.1)

The `XkbSymInterpretRec` structure specifies a symbol interpretation:

```
typedef struct {
    KeySym      sym;          /* keysym of interest or NULL */
    unsigned char flags;       /* XkbSI_AutoRepeat, XkbSI_LockingKey */
    unsigned char match;      /* specifies how mods is interpreted */
    unsigned char mods;       /* modifier bits, correspond to 8 real modifiers */
    unsigned char virtual_mod; /* 1 modifier to add to key virtual mod map */
    XkbAnyAction act;         /* action to bind to symbol position on key */
} XkbSymInterpretRec,*XkbSymInterpretPtr;
```

If `sym` is not `NULL`, it limits the symbol interpretation to keys on which that particular key-sym is selected by the modifiers matching the criteria specified by `mods` and `match`. If `sym` is `NULL`, the interpretation may be applied to any symbol selected on a key when the modifiers match the criteria specified by `mods` and `match`.

`match` must be one of the values shown in Table 17.1, and specifies how the real modifiers specified in `mods` are to be interpreted.

Table 17.1 Symbol Interpretation Match Criteria

<i>Match Criteria</i>	<i>Value</i>	<i>Effect</i>
<code>XkbSI_NonOf</code>	(0)	None of the bits which are on in <code>mods</code> can be set, but other bits can be.
<code>XkbSI_AnyOfOrNone</code>	(1)	Zero or more of the bits which are on in <code>mods</code> can be set, as well as others.
<code>XkbSI_AnyOf</code>	(2)	One or more of the bits which are on in <code>mods</code> can be set, as well as any others.
<code>XkbSI_Allof</code>	(3)	All of the bits which are on in <code>mods</code> must be set, but others may be set as well.
<code>XkbSI_Exactl</code>	(4)	All of the bits which are on in <code>mods</code> must be set, and no other bits may be set.

In addition to the above bits, `match` may contain the `XkbSI_LevelOneOnly` bit, in which case the modifier match criteria specified by `mods` and `match` applies only if `sym` is in level

one of its group; otherwise, *mods* and *match* are ignored and the symbol matches a condition where no modifiers are set.

```
#define XkbSI_LevelOneOnly (0x80) /* use mods + match only if sym is level 1 */
```

If no matching symbol interpretation is found, the server uses a default interpretation where:

<i>sym</i> =	0
<i>flags</i> =	XkbSI_AutoRepeat
<i>match</i> =	XkbSI_AnyOfOrNone
<i>mods</i> =	0
<i>virtual_mod</i> =	XkbNoModifier
<i>act</i> =	SA_NoAction

When a matching symbol interpretation is found in step 2a, the interpretation is applied to modify the Xkb map as follows:

The *act* field specifies a single action to be bound to the symbol position; any key event which selects the symbol causes the action to be taken. Valid actions are defined in section 16.1.

If the Xkb keyboard map for the key does not have its `ExplicitVModMap` control set, the `XkbSI_LevelOneOnly` bit and symbol position are examined. If the `XkbSI_LevelOneOnly` bit is not set in *match* or the symbol is in position G1L1, the *virtual_mod* field is examined. If *virtual_mod* is not `XkbNoModifier`, *virtual_mod* specifies a single virtual modifier to be added to the virtual modifier map for the key. *virtual_mod* is specified as an index in the range [0..15].

If the matching symbol is in position G1L1 of the key, two bits in the *flags* field potentially specify additional behavior modifications:

```
#define XkbSI_AutoRepeat (1<<0) /* key repeats if sym is in position G1L1 */
#define XkbSI_LockingKey (1<<1) /* set KB_Lock behavior if sym is in psn G1L1 */
```

If the Xkb keyboard map for the key does not have its `ExplicitAutoRepeat` control set, its auto repeat behavior is set based on the value of the `XkbSI_AutoRepeat` bit. If the `XkbSI_AutoRepeat` bit is set, the auto-repeat behavior of the key is turned on; otherwise, it is turned off.

If the Xkb keyboard map for the key does not have its `ExplicitBehavior` control set, its locking behavior is set based on the value of the `XkbSI_LockingKey` bit. If `XkbSI_LockingKey` is set, the key behavior is set to `KB_Lock`; otherwise, it is turned off (see section 16.3).

17.1.3 Xkb Keyboard Mapping to Core Keyboard Mapping Transformations

Whenever the server processes Xkb requests to change the keyboard mapping, it discards the affected portion of its core keyboard mapping and regenerates it based on the new Xkb mapping.

When the Xkb mapping for a key is transformed to a core protocol mapping, the symbols for the core map are taken in the following order from the Xkb map:

G1L1 G1L2 G2L1 G2L2 G1L3-n G2L3-n G3L1-n G4L1-n

If group one is of width one in the Xkb map, G1L2 is taken to be NoSymbol; similarly, if group two is of width one in the Xkb map, G2L2 is taken to be NoSymbol.

If the Xkb key map for a particular key has fewer groups than the core keyboard, the symbols for group one are repeated to fill in the missing core components. For example, an Xkb key with a single width-three group would be mapped to a core mapping counting three groups as:

G1L1 G1L2 G1L1 G1L2 G1L3 G1L3 G1L1 G1L2 G1L3

When a core keyboard map entry is generated from an Xkb keyboard map entry, a modifier mapping is generated as well. The modifier mapping contains all of the modifiers affected by any of the actions associated with the key combined with all of the real modifiers associated with any of the virtual modifiers bound to the key. In addition, if any of the actions associated with the key affect any component of the keyboard group, all of the modifiers in the *mask* field of all of the group compatibility maps are added to the modifier mapping as well. While an XkbSA_ISOLock action can theoretically affect any modifier, if the Xkb mapping for a key specifies an XkbSA_ISOLock action, only the modifiers or group that are set by default are added to the modifier mapping.

17.2 Getting Compatibility Map Components From the Server

Use *XkbGetCompatMap* to fetch any combination of the current compatibility map components from the server. When another client modifies the compatibility map, you are notified if you have selected for *XkbCompatMapNotify* events (see section 17.5). *XkbGetCompatMap* is particularly useful when you receive an event of this type, as it allows you to update your program's version of the compatibility map to match the modified version now in the server. If your program is dealing with multiple servers and needs to configure them all in a similar manner, the updated compatibility map may be used to reconfigure other servers.

Note To make a complete matching configuration you must also update the explicit override components of the server state.

Status **XkbGetCompatMap**(*display*, *which*, *xkb*)

Display *	<i>display</i> ;	/* connection to server */
unsigned int	<i>which</i> ;	/* mask of compatibility map components to fetch */
XkbDescRec *	<i>xkb</i> ;	/* keyboard description where results placed */

XkbGetCompatMap fetches the components of the compatibility map specified in *which* from the server specified by *display* and places them in the *compat* structure of the keyboard description *xkb*. Valid values for *which* are an inclusive OR of the values shown in Table 17.2.

Table 17.2 Compatibility Map Component Masks

<i>Mask</i>	<i>Value</i>	<i>Affecting</i>
XkbSymInterpMask	(1<<0)	Symbol interpretations
XkbGroupCompatMask	(1<<1)	Group maps
XkbAllCompatMask	(0x3)	All compatibility map components

If no compatibility map structure is allocated in *xkb* upon entry, *XkbGetCompatMap* allocates one. If one already exists, its contents are overwritten with the returned results.

XkbGetCompatMap fetches compatibility map information for the device specified by the *device_spec* field of *xkb*. Unless you have specifically modified this field, it is the default keyboard device. *XkbGetCompatMap* returns Success if successful, BadAlloc if it is unable to obtain necessary storage for either the return values or work space, BadMatch if the *dpy* field of the *xkb* argument is non-NULL and does not match the *display* argument, and BadLength under certain conditions caused by server or Xkb implementation errors.

17.3 Using the Compatibility Map

Xkb provides several functions which make it easier to apply the compatibility map to configure a client side Xkb keyboard mapping, given a core protocol representation of part or all of a keyboard mapping. Obtain a core protocol representation of a keyboard mapping from an actual server (by using *XGetKeyboardMapping*, for example), a data file, or some other source.

To update a local Xkb keyboard map to reflect the mapping expressed by a core format mapping by calling the function *XkbUpdateMapFromCore*.

```
Bool XkbUpdateMapFromCore(xkb, first_key, num_keys, map_width, core_keysyms, changes)
    XkbDescPtr      xkb;          /* keyboard description to update */
    KeyCode         first_key;      /* keycode of first key description to update */
    int             num_keys;      /* number of key descriptions to update */
    int             map_width;     /* width of core protocol keymap */
    KeySym *        core_keysyms;  /* symbols in core protocol keymap */
    XkbChangesPtr   changes;       /* backfilled with changes made to Xkb */
```

XkbUpdateMapFromCore interprets input argument information representing a keyboard map in core format to update the Xkb keyboard description passed in *xkb*. Only a portion of the Xkb map is updated — the portion corresponding to keys with keycodes in the range *first_key* through *first_key* + *num_keys* - 1. If *XkbUpdateMapFromCore* is being called in response to a *MappingNotify* event, *first_key* and *num_keys* are reported in the *MappingNotify* event. *core_keysyms* contains the keysyms corresponding to the keycode range being updated, in core keyboard description order. *map_width* is the number of keysyms per key in *core_keysyms*. Thus, the first *map_width* entries in *core_keysyms* are for the key with keycode *first_key*, the next *map_width* entries are for key *first_key* + 1, etc.

In addition to modifying the Xkb keyboard mapping in *xkb*, *XkbUpdateMapFromCore* backfills the changes structure whose address is passed in *changes* to indicate the modifications which were made. You may then use *changes* in subsequent calls such as *XkbSetMap*, to propagate the local modifications to a server.

When dealing with core keyboard mappings or descriptions, it is sometimes necessary to determine the Xkb key types appropriate for the symbols bound to a key in a core keyboard mapping. Call *XkbKeyTypesForCoreSymbols* for this purpose:

```
int XkbKeyTypesForCoreSymbols(map_width, core_syms, protected, types_inout,
                               xkb_syms_rtrn)
    XkbDescPtr xkb;          /* keyboard description in which to place symbols*/
    int         map_width;   /* width of core protocol keymap in xkb_syms_rtrn */
    KeySym *   core_syms;   /* core protocol format array of KeySyms */
    unsigned int protected;  /* explicit key types */
    int *      types_inout; /* backfilled with the canonical types bound to groups one and
                           two for the key. */
    KeySym *   xkb_syms_rtrn; /* backfilled with symbols bound to the key in the Xkb
                           mapping. */
```

XkbKeyTypesForCoreSymbols expands the symbols in *core_syms* and types in *types_inout* according to the rules specified in section 12 of the core protocol, then chooses canonical key types (canonical key types are defined in section 15.2.1 on page 129) for groups 1 and 2 using the rules specified by the Xkb protocol, and places them in *xkb_syms_rtrn*, which will be non-NULL.

A core keymap is a two-dimensional array of keysyms. It has *map_width* columns and *max_key_code* rows. *XkbKeyTypesForCoreSymbols* takes a single row from a core keymap, determines the number of groups associated with it, the type of each group, and the symbols bound to each group. The return value is the number of groups, *types_inout* has the types for each group, and *xkb_syms_rtrn* has the symbols in Xkb order (that is, groups are contiguous, regardless of size).

protected contains the explicitly protected key types. There is one explicit override control associated with each of the four possible groups for each Xkb key, *ExplicitKeyType1* through *ExplicitKeyType4*; *protected* is an inclusive OR of these controls. *map_width* is the width of the core keymap and isn't dependent on any Xkb definitions. *types_inout* is an array of four type indices. On input, *types_inout* contains the indices of any types already assigned to the key, in case they're explicitly protected from change.

Upon return, *types_inout* contains any automatically selected (that is, canonical) types plus any protected types. Canonical types are assigned to all four groups if there are enough symbols to do so. The four entries in *types_inout* correspond to the four groups for the key in question.

If the groups mapping does not change, but the symbols assigned to an Xkb keyboard compatibility map do change, the semantics of the key may be modified. To apply the new compatibility mapping to an individual key to get its semantics updated, call *XkbApplyCompatMapToKey*.

```
Bool XkbApplyCompatMapToKey(xkb, key, changes)
    XkbDescPtr xkb;          /* keyboard description to be updated */
    KeyCode     key;          /* key to be updated */
    XkbChangesPtr changes;   /* notes changes to the Xkb keyboard description */
```

XkbApplyCompatMapToKey essentially performs the operation described in section 17.1.2 to a specific key. This updates the behavior, actions, repeat status and virtual modifier bindings of the key.

17.4 Changing the Server's Compatibility Map

To modify the server's compatibility map, first modify a local copy of the Xkb compatibility map, then call *XkbSetCompatMap*. You may allocate a new compatibility map for this purpose using *XkbAllocCompatMap* (see section 17.6). You may also use a compatibility map from another server, although you need to adjust the *device_spec* field in the *XkbDescRec* accordingly. Note that symbol interpretations in a compatibility map (*sym_interpret*, the vector of *XkbSymInterpretRec* structures) are also allocated using this same function.

```
Bool XkbSetCompatMap(display, which, xkb, update_actions)
    Display *          display;           /* connection to server */
    unsigned int       which;            /* mask of compat map components to set */
    XkbDescPtr        xkb;             /* source for compat map components */
    Bool              update_actions; /* True => apply to server's keyboard map */
```

XkbSetCompatMap copies compatibility map information from the keyboard description in *xkb* to the server specified in *display*'s compatibility map for the device specified by the *device_spec* field of *xkb*. Unless you have specifically modified this field, it is the default keyboard device. *which* specifies the compatibility map components to be set, and is an inclusive OR of the bits shown in Table 17.2 on page 174.

After updating its compatibility map for the specified device, if *update_actions* is True the server applies the new compatibility map to its entire keyboard for the device to generate a new set of key semantics, compatibility state, and a new core keyboard map. If *update_actions* is False, the new compatibility map is not used to generate any modifications to the current device semantics, state, or core keyboard map. One reason for not applying the compatibility map immediately would be if one server was being configured to match another on a piecemeal basis; the map shouldn't be applied until everything is updated. To force an update at a later point in time, call *XkbSetCompatMap* specifying *which* as zero and *update_actions* as True.

XkbSetCompatMap returns True if successful, and False if unsuccessful. The server may report problems it encounters when processing the request subsequently via protocol errors.

To add a symbol interpretation to the list of symbol interpretations in an *XkbCompatRec*, call *XkbAddSymInterpret*.

```
XkbSymInterpretPtr XkbAddSymInterpret(xkb, si, updateMap, changes)
    XkbDescPtr        xkb;           /* keyboard description to be updated */
    XkbSymInterpretPtr si;            /* symbol interpretation to be added */
    Bool              updateMap; /* True=>apply compatibility map to keys */
    XkbChangesPtr     changes; /* changes are put here */
```

XkbAddSymInterpret adds *si* to the list of symbol interpretations in *xkb*. If *updateMap* is True, it (re)applies the compatibility map to all of the keys on the keyboard. If *changes* is non-NULL, it reports the parts of the keyboard that were affected (unless *updateMap* is True, not much changes). *XkbAddSymInterpret* returns a pointer to the actual new symbol interpretation in the list, or NULL if it failed.

17.5 Tracking Changes to the Compatibility Map

The server automatically generates `MappingNotify` events when the keyboard mapping changes. If you wish to be notified of changes to the compatibility map, you should select for `XkbCompatMapNotify` events. If you select for `XkbMapNotify` events, you no longer receive the automatically generated `MappingNotify` events. If you subsequently deselect `XkbMapNotifyEvent` delivery, you again receive `MappingNotify` events.

To receive `XkbCompatMapNotify` events under all possible conditions, call `XkbSelectEvents` (see section 4.3) and pass `XkbCompatMapNotifyMask` in both `bits_to_change` and `values_for_bits`.

To receive `XkbCompatMapNotify` events only under certain conditions, call `XkbSelectEventDetails` using `XkbCompatMapNotify` as the `event_type` and specifying the desired map changes in `bits_to_change` and `values_for_bits` using mask bits from Table 17.2 on page 174.

Note that you are notified of changes you make yourself, as well as changes made by other clients.

The structure for the `XkbCompatMapNotifyEvent` is:

```
typedef struct {
    int             type;          /* Xkb extension base event code */
    unsigned long   serial;        /* X server serial number for event */
    Bool            send_event;    /* True => synthetically generated */
    Display *      display;       /* server connection where event generated */
    Time           time;          /* server time when event generated */
    int             xkb_type;     /* XkbCompatMapNotify */
    int             device;        /* Xkb device id, will not be XkbUseCoreKbd */
    unsigned int    changed_groups; /* number of group maps changed */
    int             first_si;      /* index to 1st changed symbol interpretation */
    int             num_si;        /* number of changed symbol interpretations */
    int             num_total_si;  /* total number of valid symbol interpretations */
} XkbCompatMapNotifyEvent;
```

`changed_groups` is the number of group compatibility maps which have changed. If you are maintaining a corresponding copy of the compatibility map, or get a fresh copy from the server using `XkbGetCompatMap`, `changed_groups` references `groups[0..changed_groups-1]` in the `XkbCompatMapRec` structure.

`first_si` is the index of the first changed symbol interpretation, `num_si` is the number of changed symbol interpretations, and `num_total_si` is the total number of valid symbol interpretations. If you are maintaining a corresponding copy of the compatibility map, or get a fresh copy from the server using `XkbGetCompatMap`, `first_si`, `num_si`, and `num_total_si` are appropriate for use with the `compat.sym_interpret` vector in this structure.

17.6 Allocating and Freeing Compatibility Map

If you are modifying the compatibility map, you need to allocate a new compatibility map if you do not already have one available. To do so, call *XkbAllocCompatMap*.

```
Status XkbAllocCompatMap(xkb, which, num_si)
    XkbDescPtr xkb;          /* keyboard description in which to allocate compat map */
    unsigned int which;      /* mask of compatibility map components to allocate */
    unsigned int num_si;     /* number of symbol interpretations to allocate */
```

xkb specifies the keyboard description for which compatibility maps are to be allocated. The compatibility map is the *compat* field in this structure.

which specifies the compatibility map components to be allocated (see *XkbGetCompatMap*, in section 17.2). *which* is an inclusive OR of the bits shown in Table 17.2 on page 174.

num_si specifies the total number of entries to allocate in the symbol interpretation vector (*xkb.compat.sym_interpret*).

Note that symbol interpretations in a compatibility map (the *sym_interpret* vector of *XkbSymInterpretRec* structures) are also allocated using this same function. To insure that there is sufficient space in the symbol interpretation vector for entries to be added, call *XkbAllocCompatMap* specifying *which* as *XkbSymInterpretMask*, and the number of free symbol interpretations needed in *num_si*.

XkbAllocCompatMap returns Success if successful, BadMatch if *xkb* is NULL, or BadAlloc if errors are encountered when attempting to allocate storage.

To free an entire compatibility map or selected portions of one, call *XkbFreeCompatMap*.

```
void XkbFreeCompatMap(xkb, which, free_map)
    XkbDescPtr xkb;          /* Xkb description in which to free compatibility map */
    unsigned int which;      /* mask of compatibility map components to free */
    Bool       free_map;     /* True => free XkbCompatMap structure itself */
```

which specifies the compatibility map components to be freed (see *XkbGetCompatMap*, in section 17.2). *which* is an inclusive OR of the bits shown in Table 17.2 on page 174.

free_map indicates whether the *XkbCompatMap* structure itself should be freed. If *free_map* is True, *which* is ignored, all non-NULL compatibility map components are freed, and the *compat* field in the *XkbDescRec* referenced by *xkb* is set to NULL.

18 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 an application that 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 then 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).

To make it easier for applications to present a keyboard to the user, Xkb supports symbolic names for most components of the keyboard extension. Most of these symbolic names are grouped into the *names* component of the keyboard description.

18.1 The XkbNamesRec Structure

The names component of the keyboard description is defined as follows:

```
#define XkbKeyNameLength      4
#define XkbKeyNumVirtualMods  16
#define XkbKeyNumIndicators   32
#define XkbKeyNumKbdGroups    4
#define XkbMaxRadioGroups     32

typedef struct {
    char          name[XkbKeyNameLength]; /* symbolic key names */
} XkbKeyNameRec,*XkbKeyNamePtr;

typedef struct {
    char          real[XkbKeyNameLength]; /* this key name must be in the keys array */
    char          alias[XkbKeyNameLength]; /* symbolic key name as alias for the key */
} XkbKeyAliasRec,*XkbKeyAliasPtr;

typedef struct _XkbNamesRec {
    Atom          keycodes;        /* identifies range and meaning of keycodes */
    Atom          geometry;       /* identifies physical location, size and shape of keys */
    Atom          symbols;        /* identifies the symbols logically bound to the keys */
    Atom          types;          /* identifies the set of key types */
    Atom          compat;         /* identifies actions for keys using core protocol */
    Atom          vmods[XkbNumVirtualMods]; /* symbolic names for virtual modifiers */
    Atom          indicators[XkbNumIndicators]; /* symbolic names for indicators */
    Atom          groups[XkbNumKbdGroups]; /* symbolic names for keyboard groups */
    XkbKeyNamePtr keys;          /* symbolic key name array */
    XkbKeyAliasPtr key_aliases;  /* real/alias symbolic name pairs array */
    Atom *        radio_groups;   /* radio group name array */
    Atom          phys_symbols;  /* identifies the symbols engraved on the keyboard */
    unsigned char num_keys;      /* number of keys in the keys array */
    unsigned char num_key_aliases; /* number of keys in the key_aliases array */
    unsigned short num_rg;       /* number of radio groups */
} XkbNamesRec,*XkbNamesPtr;*/
```

The *keycodes* name identifies the range and meaning of the keycodes returned by the keyboard in question. The *geometry* name, on the other hand, identifies 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 share a similar `keycodes` name but may have 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 all keys have the same semantics when present. For example, 101 and 102 key PC keyboards might use the same name. In these cases, applications can use the keyboard `geometry` name to determine which subset of the named keycodes is in use.

The `symbols` name identifies the symbols logically 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 `phys_symbols` name, on the other hand, identifies the symbols actually engraved on the keyboard. Given this, the `symbols` name and `phys_symbols` names might be different. For example, the description for a keyboard that has English US engravings, but that is using Swiss German symbols might have a `phys_symbols` name of “en_US” and a `symbols` name of “de_CH.”

The `types` name provides some information about the set of key types (see section 15.2) that can be associated with the keyboard. In addition, each key type can have a name, and each shift level of a type can have a name. Although these names are stored in the map description with each of the types, they are accessed using the same methods as the other symbolic names.

The `compat` name provides some information about the rules used to bind actions to keys that are changed using core protocol requests.

Xkb provides symbolic names for each of the four keyboard groups, sixteen virtual modifiers, thirty-two keyboard indicators, and four keyboard groups. These names are held in the `vmods`, `indicators`, and `groups` fixed-length arrays.

Each key has a four-byte symbolic name. All of the symbolic key names are held in the `keys` array, and `num_keys` reports the number of entries are in the `keys` array. For each key, the key name links keys with similar functions or in similar positions on keyboards that report different keycodes. For example, the F1 key may emit keycode 23 on one keyboard and keycode 86 on another. By naming this key “FK01” on both keyboards, the keyboard layout designer can re-use parts of keyboard descriptions for different keyboards.

Key aliases allow the keyboard layout designer to assign multiple key names to a single key. This allows the keyboard layout designer to refer to keys using either their position or their “function.” For example, a keyboard layout designer may wish to refer to the left arrow key on a PC keyboard using the ISO9995-5 positional specification of A31 or using the functional specification of LEFT. The `key_aliases` field holds a variable-length array of real and alias key name pairs, and the total number of entries in the `key_aliases` array is held in `num_key_aliases`. For each real and alias key name pair, the `real` field refers to the a name in the `keys` array, and the `alias` field refers to the alias for that key. Using the previous example, the keyboard designer may use the name A31 in the `keys` array, but also define the name LEFT as an alias for A31 in the `key_aliases` array.

Note Key aliases defined in the geometry component of a keyboard mapping (see Chapter 13) override those defined in the keycodes component of the server database, which are stored in the `XkbNamesRec` (`xkb->names`). Therefore, consider the key aliases defined by the geometry before considering key aliases supplied by the `XkbNamesRec`.

A radio group is a set of keys whose behavior simulates a set of radio buttons. Once a key in a radio group is pressed, it stays logically depressed until another key in the group is pressed, at which point the previously depressed key is logically released. Consequently, at most one key in a radio group can be logically depressed at one time.

Each radio group in the keyboard description can have a name. These names are held in the variable-length array `radio_groups`, and `num_rg` tells how many elements are in the `radio_groups` array.

18.2 Symbolic Names Masks

Xkb provides several routines that work with symbolic names. Each of these routines uses a mask to specify individual fields of the structures described above. These masks and their relationships to the fields in a keyboard description are shown in Table 18.1.

Table 18.1 Symbolic Names Masks

<i>Mask Bit</i>	<i>Value</i>	<i>Keyboard Component</i>	<i>Field</i>
<code>XkbKeycodesNameMask</code>	(1<<0)	<code>Xkb->names</code>	<code>keycodes</code>
<code>XkbGeometryNameMask</code>	(1<<1)	<code>Xkb->names</code>	<code>geometry</code>
<code>XkbSymbolsNameMask</code>	(1<<2)	<code>Xkb->names</code>	<code>symbols</code>
<code>XkbPhysSymbolsNameMask</code>	(1<<3)	<code>Xkb->names</code>	<code>phys_symbols</code>
<code>XkbTypesNameMask</code>	(1<<4)	<code>Xkb->names</code>	<code>type</code>
<code>XkbCompatNameMask</code>	(1<<5)	<code>Xkb->names</code>	<code>compat</code>
<code>XkbKeyTypeNamesMask</code>	(1<<6)	<code>Xkb->map</code>	<code>type[*].name</code>
<code>XkbKTLvlNamesMask</code>	(1<<7)	<code>Xkb->map</code>	<code>type[*].lvl_names[*]</code>
<code>XkbIndicatorNamesMask</code>	(1<<8)	<code>Xkb->names</code>	<code>indicators[*]</code>
<code>XkbKeyNamesMask</code>	(1<<9)	<code>Xkb->names</code>	<code>keys[*], num_keys</code>
<code>XkbKeyAliasesMask</code>	(1<<10)	<code>Xkb->names</code>	<code>key_aliases[*], num_key_aliases</code>
<code>XkbVirtualModNamesMask</code>	(1<<11)	<code>Xkb->names</code>	<code>vmods[*]</code>
<code>XkbGroupNamesMask</code>	(1<<12)	<code>Xkb->names</code>	<code>groups[*]</code>
<code>XkbRGNamesMask</code>	(1<<13)	<code>Xkb->names</code>	<code>radio_groups[*], num_rg</code>
<code>XkbComponentNamesMask</code>	(0x3ff)	<code>Xkb->names</code>	<code>keycodes, geometry, symbols, physical symbols, types, and compatibility map</code>
<code>XkbAllNamesMask</code>	(0x3fff)	<code>Xkb->names</code>	<code>all name components</code>

18.3 Getting Symbolic Names From the Server

To obtain symbolic names from the server, call *XkbGetNames*.

```
Status XkbGetNames(dpy, which, Xkb)
    Display * dpy;      /* connection to the X server */
    unsigned int which; /* mask of names or map components to be updated */
    XkbDescPtr xkb;    /* keyboard description to be updated */
```

XkbGetNames retrieves symbolic names for the components of the keyboard extension from the X server. The *which* parameter specifies the name components to be updated in the *xkb* parameter, and is the bitwise inclusive OR of the valid names mask bits defined in Table 18.1.

If the *names* field of the keyboard description *xkb* is NULL, *XkbGetNames* allocates and initializes the *names* component of the keyboard description before obtaining the values specified by *which*. If the *names* field of *xkb* is not NULL, *XkbGetNames* obtains the values specified by *which* and copies them into the keyboard description *Xkb*.

If the *map* component of the *xkb* parameter is NULL, *XkbGetNames* does not retrieve type or shift level names, even if *XkbKeyTypeNamesMask* or *XkbKTLLevelNamesMask* are set in *which*.

XkbGetNames can return Success, or BadAlloc, BadLength, BadMatch, and BadImplementation errors.

To free symbolic names, use *XkbFreeNames* (see section 18.6)

18.4 Changing Symbolic Names on the Server

To change the symbolic names in the server, first modify a local copy of the keyboard description and then use either *XkbSetNames* or, to save network traffic, use a *XkbNameChangesRec* structure and call *XkbChangeNames* to download the changes to the server. *XkbSetNames* and *XkbChangeNames* can generate BadAlloc, BadAtom, BadLength, BadMatch and BadImplementation errors.

```
Bool XkbSetNames(dpy, which, first_type, num_types, xkb)
    Display * dpy;      /* connection to the X server */
    unsigned int which; /* mask of names or map components to be changed */
    unsigned int first_type; /* first type whose name is to be changed */
    unsigned int num_types; /* number of types for which names are to be changed */
    XkbDescPtr xkb;    /* keyboard description from which names are to be taken */
```

Use *XkbSetNames* to change many names at the same time. For each bit set in *which*, *XkbSetNames* takes the corresponding value (or values in the case of arrays) from the keyboard description *xkb* and sends it to the server.

The *first_type* and *num_types* arguments are used only if *XkbKeyTypeNamesMask* or *XkbKTLLevelNamesMask* is set in *which*, and specify a subset of the types for which the corresponding names are to be changed. If either or both of these mask bits are set but the specified types are illegal, *XkbSetNames* returns False and does not update any of the names specified in *which*. The specified types are illegal if *xkb* does not include a map component or if *first_type* and *num_types* specify types that are not defined in the keyboard description.

The XkbNameChangesRec Structure

The XkbNameChangesRec allows applications to identify small modifications to the symbolic names and effectively reduces the amount of traffic sent to the server:

```
typedef struct _XkbNameChanges {
    unsigned int    changed;          /* name components that have changed */
    unsigned char   first_type;       /* first key type with a new name */
    unsigned char   num_types;        /* number of types with new names */
    unsigned char   first_lvl;        /* first key type with new level names */
    unsigned char   num_lvls;         /* number of key types with new level names */
    unsigned char   num_aliases;      /* if key aliases changed, total number of key aliases */
    unsigned char   num_rg;           /* if radio groups changed, total number of radio groups */
    unsigned char   first_key;        /* first key with a new name */
    unsigned char   num_keys;         /* number of keys with new names */
    unsigned short  changed_vmods;    /* mask of virtual modifiers for which names have changed */
    unsigned long   changed_indicators; /* mask of indicators for which names were changed */
    unsigned char   changed_groups;   /* mask of groups for which names were changed */
} XkbNameChangesRec, *XkbNameChangesPtr
```

The *changed* field specifies the name components that have changed and is the bitwise inclusive OR of the valid names mask bits defined in Table 18.1 on page 182. The rest of the fields in the structure specify the ranges that have changed for the various kinds of symbolic names, as shown in Table 18.2.

Table 18.2 XkbNameChanges Fields

Mask	Fields	Component	Field
XkbKeyTypeNamesMask	first_type, num_types	Xkb->map	type[*].name
XkbKTLevelNamesMask	first_lvl, num_lvls	Xkb->map	type[*].lvl_names[*]
XkbKeyAliasesMask	num_aliases	Xkb->names	key_aliases[*]
XkbRGNamesMask	num_rg	Xkb->names	radio_groups[*]
XkbKeyNamesMask	first_key, num_keys	Xkb->names	keys[*]
XkbVirtualModNamesMask	changed_vmods	Xkb->names	vmods[*]
XkbIndicatorNamesMask	changed_indicators	Xkb->names	indicators[*]
XkbGroupNamesMask	changed_groups	Xkb->names	groups[*]

XkbChangeNames provides a more flexible method for changing symbolic names than *XkbSetNames*, and requires the use of an XkbNameChangesRec structure.

Bool **XkbChangeNames**(*dpy*, *xkb*, *changes*)

```
Display *          dpy;      /* connection to the X server */
XkbDescPtr        xkb;      /* keyboard description from which names are to be taken */
XkbNameChangesPtr changes;  /* names map components to be updated on the server */
```

XkbChangeNames copies any names specified by *changes* from the keyboard description, *xkb*, to the X server specified by *dpy*. *XkbChangeNames* aborts and returns False if any illegal type names or type shift level names are specified by *changes*.

18.5 Tracking Name Changes

Whenever a symbolic name changes in the server's keyboard description, the server sends a `XkbNamesNotify` event to all interested clients. To receive name notify events, call `XkbSelectEvents` (see section 4.3) with `XkbNamesNotifyMask` in both the `bits_to_change` and `values_for_bits` parameters.

To receive events for only specific names, use `XkbSelectEventDetails`. Set the `event_type` parameter to `XkbNamesNotify`, and set both the `bits_to_change` and `values_for_bits` detail parameter to a mask composed of a bitwise OR of masks in Table 18.1 on page 182.

The structure for the `XkbNamesNotify` event is defined as follows:

```
typedef struct {
    int          type;           /* Xkb extension base event code */
    unsigned long serial;       /* X server serial number for event */
    Bool         send_event;     /* True => synthetically generated */
    Display *   display;        /* server connection where event generated */
    Time         time;           /* server time when event generated */
    int          xkb_type;       /* XkbNamesNotify */
    int          device;         /* Xkb device id, will not be XkbUseCoreKbd */
    unsigned int changed;       /* mask of name components that have changed */
    int          first_type;     /* first key type with a new name */
    int          num_types;      /* number of types with new names */
    int          first_lvl;      /* first key type with new level names */
    int          num_lvls;       /* number of key types with new level names */
    int          num_aliases;    /* if key aliases changed, total number of key aliases */
    int          num_radio_groups; /* if radio groups changed, total number of radio groups */
    unsigned int changed_vmods; /* mask of virtual modifiers for which names have changed */
    unsigned int changed_groups; /* mask of groups for which names were changed */
    unsigned int changed_indicators; /* mask of indicators for which names were changed */
    int          first_key;      /* first key with a new name */
    int          num_keys;       /* number of keys with new names */
} XkbNamesNotifyEvent;
```

The `changed` field specifies the name components that have changed and is the bitwise inclusive OR of the valid names mask bits defined in Table 18.1 on page 182. The other fields in this event are interpreted as the like-named fields in an `XkbNameChangesRec`, as defined above.

When your application receives a `XkbNamesNotify` event, you can note the changed names in a changes structure using `XkbNoteNameChanges`.

```
void XkbNoteNameChanges(old, new, wanted)
XkbNameChangesPtr old;           /* XkbNameChanges structure to be updated */
XkbNamesNotifyEvent *new;         /* event from which changes are to be copied */
unsigned int         wanted;     /* types of names for which changes are to be noted */
```

The `wanted` parameter is the bitwise inclusive OR of the valid names mask bits shown in Table 18.1 on page 182. `XkbNoteNameChanges` copies any changes that are reported in `new` and specified in `wanted` into the changes record specified by `old`.

To update the local copy of the keyboard description with the actual values, pass to *XkbGetNameChanges* the results of one or more calls to *XkbNoteNameChanges*.

```
Status XkbGetNameChanges(dpy, xkb, changes)
    Display *          dpy;      /* connection to the X server */
    XkbDescPtr        xkb;      /* keyboard description to which names are copied */
    XkbNameChangesPtr changes;   /* names components to be obtained from the server */
```

XkbGetNameChanges examines the *changes* parameter, retrieves the necessary information from the server, and places the results into the *xkb* keyboard description.

XkbGetNamesChanges can generate BadAlloc, BadImplementation and BadMatch errors.

18.6 Allocating and Freeing Symbolic Names

Most applications do not need to directly allocate symbolic names structures. Do not allocate a names structure directly using *malloc* or *Xmalloc* if your application changes the number of key aliases or radio groups or constructs a symbolic names structure without loading the necessary components from the X server. Instead use *XkbAllocNames*.

```
Status XkbAllocNames(xkb, which, num_rg, num_key_aliases)
    XkbDescPtr xkb;      /* keyboard description for which names are to be allocated */
    unsigned int which;    /* mask of names to be allocated */
    int         num_rg;   /* total number of radio group names needed */
    int         num_key_aliases; /* total number of key aliases needed */
```

XkbAllocNames can return BadAlloc, BadMatch and BadValue errors. The *which* parameter is the bitwise inclusive OR of the valid names mask bits defined in Table 18.1 on page 182.

Do not free symbolic names structures directly using *free* or *XFree*. Use *XkbFreeNames* instead.

```
void XkbFreeNames(xkb, which, free_map)
    XkbDescPtr xkb;      /* keyboard description for which names are to be freed */
    unsigned int which;    /* mask of names components to be freed */
    Bool       free_map;  /* True => XkbNamesRec structure itself should be freed */
```

The *which* parameter is the bitwise inclusive OR of the valid names mask bits defined in Table 18.1 on page 182.

19 Replacing a Keyboard “On the Fly”

Some operating system and X server implementations allow “hot-plugging” of input devices. When using these implementations, input devices can be unplugged and new ones plugged in without restarting the software that is using those devices. There is no provision in the standard X server for notification of client programs if input devices are unplugged and/or new ones plugged in. In the case of the X keyboard, this could result in the X server having a keymap that does not match the new keyboard.

If the X server implementation supports the X input device extension, a client program may also change the X keyboard programmatically. The *XChangeKeyboardDevice* input extension request allows a client to designate an input extension keyboard device as the X keyboard, in which case the old X keyboard device becomes inaccessible except via the input device extension. In this case, core protocol *XMappingNotify* and input extension *XChangeDeviceNotify* events are generated to notify all clients that a new keyboard with a new keymap has been designated.

When a client opens a connection to the X server, the server reports the minimum and maximum keycodes. The server keeps track of the minimum and maximum keycodes last reported to each client. When delivering events to a particular client, the server filters out any events that fall outside of the valid range for the client.

Xkb provides an *XkbNewKeyboardNotify* event which reports a change in keyboard geometry and/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 which loads a new keyboard description. Selecting for *XkbNewKeyboardNotify* events allows Xkb-aware clients to be notified whenever a keyboard change occurs that may affect the keymap.

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 the client an *XkbNewKeyboardNotify* event. Even if the “new” keyboard is not new to the server, it is new to this particular client.

When the server sends an *XkbNewKeyboardNotify* event to a client to inform it of a new keycode range, it resets the stored range of legal keycodes for the client to the keycode range reported in the event; it does not reset this range for the client if it does not send an *XkbNewKeyboardNotify* event to a client. Since Xkb-unaware clients and Xkb-aware clients that do not request *XkbNewKeyboardNotify* events are never sent these events, the server’s notion of the legal keycode range never changes, and these clients never receive events from keys that fall outside of their notion of the legal keycode range.

Clients which have not selected to receive *XkbNewKeyboardNotify* events do, however, receive the *XkbNewKeyboardNotify* event when a keyboard change occurs. Clients which have not selected to receive this event also receive numerous other events detailing the individual changes which occur when a keyboard change occurs.

Clients wishing to track changes in *min_key_code* and *max_key_code* must watch for both *XkbNewKeyboardNotify* and *XkbMapNotify* events, since a simple mapping change causes an *XkbMapNotify* event and may change the range of valid keycodes, but does

not cause an `XkbNewKeyboardNotify` event. If a client does not select for `XkbNewKeyboardNotify` events, the server restricts the range of keycodes reported to the 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 from Xkb-unaware clients the fact that the range of legal keycodes has changed, since such clients cannot be expected to deal with them. Xkb events and requests are not modified in this manner; all Xkb events report the full range of legal keycodes. No requested Xkb events are discarded, and no Xkb requests have their keycode range clamped.

The structure for the `XkbNewKeyboardNotify` event is defined below:

```
typedef struct _XkbNewKeyboardNotify {
    int             type;           /* Xkb extension base event code */
    unsigned long   serial;         /* X server serial number for event */
    Bool            send_event;     /* True => synthetically generated */
    Display *       display;        /* server connection where event generated */
    Time            time;           /* server time when event generated */
    int             xkb_type;       /* XkbNewKeyboardNotify */
    int             device;          /* device id of new keyboard */
    int             old_device;     /* device id of old keyboard */
    int             min_key_code;   /* min keycode of new keyboard */
    int             max_key_code;   /* max keycode of new keyboard */
    int             old_min_key_code; /* min keycode of old keyboard */
    int             old_max_key_code; /* max keycode of old keyboard */
    unsigned int    changed;        /* changed aspects - see masks below */
    char            req_major;      /* major request that caused change */
    char            req_minor;      /* minor request that caused change */
} XkbNewKeyboardNotifyEvent;
```

To receive name notify events, call `XkbSelectEvents` (see section 4.3) with `XkbNewKeyboardNotifyMask` in both the `bits_to_change` and `values_for_bits` parameters. To receive events for only specific names, use `XkbSelectEventDetails`. Set the `event_type` parameter to `XkbNewKeyboardNotify`, and set both the `bits_to_change` and `values_for_bits` detail parameter to a mask composed of a bitwise OR of masks in Table 19.1.

Table 19.1 XkbNewKeyboardNotifyEvent Details

<i>XkbNewKeyboardNotify Event Details</i>	<i>Value</i>	<i>Circumstances</i>
<code>XkbNKN_KeycodesMask</code>	<code>(1L<<0)</code>	Notification of keycode range changes wanted
<code>XkbNKN_GeometryMask</code>	<code>(1L<<1)</code>	Notification of geometry changes wanted
<code>XkbNKN_DeviceIDMask</code>	<code>(1L<<2)</code>	Notification of device id changes wanted

Table 19.1 XkbNewKeyboardNotifyEvent Details

<i>XkbNewKeyboardNotify Event Details</i>	<i>Value</i>	<i>Circumstances</i>
XkbNKN_AllChangesMask	(0x7)	Includes all of the above masks

The *req_major* and *req_minor* fields indicate what type of keyboard change has occurred.

If *req_major* and *req_minor* are zero, the device change was not caused by a software request to the server — a spontaneous change has occurred such as hot-plugging a new device. In this case, *device* is the device identifier for the new, current X keyboard device, but no implementation-independent guarantee can be made about *old_device*. *old_device* may be identical to *device* (an implementor is permitted to reuse the device specifier when the device changes); or it may be different. Note that *req_major* and *req_minor* being zero do not necessarily mean that the physical keyboard device has changed; rather, they only imply a spontaneous change outside of software control (Some systems have keyboards which can change personality at the press of a key.).

If the keyboard change is the result of an X Input Extension ChangeKeyboardDevice request, *req_major* contains the input extension major opcode, and *req_minor* contains the input extension request number for X_ChangeKeyboardDevice. In this case, *device* and *old_device* are different, with *device* being the identifier for the new, current X keyboard device, and *old_device* being the identifier for the former device.

If the keyboard change is the result of an XkbGetKeyboardByName function call, which generates an X_kbGetKbdByName request, *req_major* contains the Xkb extension base event code (see section 2.4), and *req_minor* contains the event code for the Xkb extension request X_kbGetKbdByName. *device* contains the device identifier for the new device, but nothing definitive can be said for *old_device*; it may be identical to *device*, or it may be different, depending on the implementation.

20 Server Database of Keyboard Components

The X server maintains a database of keyboard components, identified by component type. The database contains all the information necessary to build a complete keyboard description for a particular device, as well as to assemble partial descriptions. Table 20.1 identifies the component types and the type of information they contain.

Table 20.1 Server Database Keyboard Components

<i>Component Type</i>	<i>Component Primary Contents</i>	<i>May also contain</i>
Keymap	Complete keyboard description Normally assembled using a complete component from each of the other types	
Keycodes	Symbolic name for each key Minimum and maximum legal keycodes	Aliases for some keys Symbolic names for indicators Description of indicators physically present
Types	Key types	Real modifier bindings and symbolic names for some virtual modifiers
Compatibility	Rules used to assign actions to keysyms	Maps for some indicators Real modifier bindings and symbolic names for some virtual modifiers
Symbols	Symbol mapping for keyboard keys Modifier mapping Symbolic names for groups	Explicit actions and behaviors for some keys Real modifier bindings and symbolic names for some virtual modifiers
Geometry	Layout of the keyboard	Aliases for some keys; overrides keycodes component aliases Symbolic names for some indicators Description of indicators physically present

While a keymap is a database entry for a complete keyboard description, and therefore logically different from the individual component database entries, the rules for processing keymap entries are identical to those for the individual components. In the discussion which follows, the term component is used to refer to either individual components or a keymap.

There may be multiple entries for each of the component types. An entry may be either *complete* or *partial*. Partial entries describe only a piece of the corresponding keyboard component, and are designed to be combined with other entries of the same type to form a complete entry.

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 be used to configure *any* ASCII keyboard to use a national layout.

When a keyboard description is built, the components are processed in the order in which they appear in Table 20.1; later definitions override earlier ones.

20.1 Component Names

Component 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)*” for a symbols component 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. Xkb places no constraints on the interpretation of the class and member names used in component names.

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.

20.2 Listing the Known Keyboard Components

You may ask the server for a list of components for one or more component types. The request takes the form of a set of patterns, one pattern for each of the component types, including a pattern for the complete keyboard description. To obtain this list, call *XkbListComponents*.

```
XkbComponentListPtr XkbListComponents(dpy, device_spec, ptrns, max_inout)
    Display *          dpy;           /* connection to X server */
    unsigned int       device_spec;   /* device id, or XkbUseCoreKbd */
    XkbComponentNamesPtr ptrns;       /* namelist for components of interest */
    int *             max_inout;     /* max # returned names, # left over */
```

XkbListComponents queries the server for a list of component names matching the patterns specified in *ptrns*. It waits for a reply and returns the matching component names in an *XkbComponentListRec* structure. When you are done using the structure you should free it using *XkbFreeComponentList*. *device_spec* indicates a particular device in which the caller is interested. A server is allowed (but not required) to restrict its reply to portions of the database which are relevant for that particular device.

ptrns is a pointer to an *XkbComponentNamesRec*, described below. Each of the fields in *ptrns* contains a pattern naming the components of interest. Each of the patterns is composed of characters from the ISO Latin1 encoding, but can contain only parentheses, the wildcard characters ‘?’ and ‘*’, and characters permitted in a component class or member name (see section 20.1). A pattern may be NULL, in which case no components for that type is returned. Pattern matches with component names are case sensitive. The ‘?’ wildcard matches any single character except a left or right parenthesis; the ‘*’ wildcard matches any number of characters except a left or right parenthesis. If an implementation allows additional characters in a component class or member name other than those required by the Xkb extension (see section 20.1), the result of comparing one of the additional characters to either of the wildcard characters is implementation dependant.

If a pattern contains illegal characters, the illegal characters are ignored. The matching process is carried out as if the illegal characters were omitted from the pattern.

max_inout is used to throttle the amount of data passed to and from the server. On input, it specifies the maximum number of names to be returned (the total number of names in all

component categories). Upon return from *XkbListComponents*, *max_inout* contains the number of names which matched the request but were not returned because of the limit.

The component name patterns used to describe the request are passed to *XkbListComponents* using an *XkbComponentNamesRec* structure. This structure has no special allocation constraints or inter-relationships with other structures, allocate and free this structure using standard *malloc* and *free* calls or their equivalent:

```
typedef struct _XkbComponentNames {
    char *keymap;          /* keymap names */
    char *keycodes;         /* keycode names */
    char *types;            /* type names */
    char *compat;           /* compatibility map names */
    char *symbols;          /* symbol names */
    char *geometry;         /* geometry names */
} XkbComponentNamesRec, *XkbComponentNamesPtr;
```

XkbListComponents returns a pointer to an *XkbComponentListRec*:

```
typedef struct _XkbComponentList {
    int num_keymaps; /* number of entries in keymap */
    int num_keycodes; /* number of entries in keycodes */
    int num_types;   /* number of entries in types */
    int num_compat;  /* number of entries in compat */
    int num_symbols; /* number of entries in symbols */
    int num_geometry; /* number of entries in geometry;
                      * keymap;      /* keymap names */
                      * keycodes;     /* keycode names */
                      * types;        /* type names */
                      * compat;       /* compatibility map names */
                      * symbols;     /* symbol names */
                      * geometry;    /* geometry names */
} XkbComponentListRec, *XkbComponentListPtr;

typedef struct _XkbComponentName {
    unsigned short flags; /* hints regarding component name */
    char *name;          /* name of component */
} XkbComponentNameRec, *XkbComponentNamePtr;
```

Note that the structure used to specify patterns on input is an *XkbComponentNamesRec*, and that used to hold the individual component names upon return is an *XkbComponentNameRec* (no trailing ‘s’ in Name).

When you are done using the structure returned by *XkbListComponents*, free it using *XkbFreeComponentList*.

```
void XkbFreeComponentList(list)
    XkbComponentListPtr list; /* pointer to XkbComponentListRec to free */
```

20.3 Component Hints

A set of flags is associated with each component; these flags provide additional hints about the component’s use. These hints are designated by bit masks in the flags field of the *XkbComponentNameRec* structures contained in the *XkbComponentListRec*

returned from `XkbListComponents`. The least significant byte of the flags field has the same meaning for all types of keyboard components; the interpretation of the most significant byte is dependent on the type of component. The flags bits are defined in Table 20.2. The symbols hints in Table 20.2 only apply to partial symbols components (those with `XkbLC_Partial` also set); full symbols components are assumed to specify all of the pieces.

The alphanumeric, modifier, keypad or function keys symbols hints should describe the primary intent of the component designer and should not be 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 as well; such mappings should set only the `XkbLC_AlphanumericKeys` hint. In general, symbols components should set only one of the four flags (`XkbLC_AlternateGroup` may be combined with any of the other flags).

Table 20.2 XkbComponentNameRec flags bits

<i>Component Type</i>	<i>Component Hints (flags)</i>	<i>Meaning</i>	<i>Value</i>
All Components	<code>XkbLC_Hidden</code>	Do not present to user	(1L<<0)
	<code>XkbLC_Default</code>	Default member of class	(1L<<1)
	<code>XkbLC_Partial</code>	Partial component	(1L<<2)
Keymap	none		
Keycodes	none		
Types	none		
Compatibility	none		
Symbols	<code>XkbLC_AlphanumericKeys</code>	Bindings primarily for alphanumeric keyboard section	(1L<<8)
	<code>XkbLC_ModifierKeys</code>	Bindings primarily for modifier keys	(1L<<9)
	<code>XkbLC_KeypadKeys</code>	Bindings primarily for numeric keypad keys	(1L<<10)
	<code>XkbLC_FunctionKeys</code>	Bindings primarily for function keys	(1L<<11)
	<code>XkbLC_AlternateGroup</code>	Bindings for an alternate group	(1L<<12)
Geometry	none		

20.4 Building a Keyboard Description using the Server Database

A client may request that the server fetch one or more components from its database, and use those components to build a new server keyboard description. The new keyboard description may be built from scratch, or it may be built starting with the current keyboard description for a particular device. Once the keyboard description is built, all or part of it may be returned to the client. The parts returned to the client need not include all of the parts used to build the description. At the time it requests the server to build a new keyboard description, a client may also request that the server use the new description internally to replace the current keyboard description for a specific device, in which case the behavior of the device changes accordingly.

To build a new keyboard description from a set of named components, and to optionally have the server use the resulting description to replace an active one, call *XkbGetKeyboardByName*.

XkbDescPtr XkbGetKeyboardByName (<i>dpy</i> , <i>device_spec</i> , <i>names</i> , <i>want</i> , <i>need</i> , <i>load</i>)	
Display *	<i>dpy</i> ; /* connection to X server */
unsigned int	<i>device_spec</i> ; /* device id, or XkbUseCoreKbd */
XkbComponentNamesPtr	<i>names</i> ; /* names of components to fetch */
unsigned int	<i>want</i> ; /* desired structures in returned record */
unsigned int	<i>need</i> ; /* mandatory structures in returned record */
Bool	<i>load</i> ; /* True => load into <i>device_spec</i> */

names contains a set of expressions describing the keyboard components the server should use to build the new keyboard description. *want* and *need* are bit fields describing the parts of the resulting keyboard description which should be present in the returned *XkbDescRec*.

The individual fields in *names* are *component expressions* composed of keyboard component names (no wildcarding as may be used in *XkbListComponents*), the special component name symbol ‘%’, and the special operator characters ‘+’ and ‘|’. A component expression is parsed left to right, as follows:

- The special component name “computed” may be used in *keycodes* component expressions, and refers to a component consisting of a set of keycodes computed automatically by the server as needed.
- The special component name “canonical” may be used in *types* component expressions, and refers to a partial component defining the four standard key types: ALPHABETIC, ONE_LEVEL, TWO_LEVEL, and KEYPAD.
- The special component name ‘%’ refers to the keyboard description for the device specified in *device_spec* or the keymap names component. If a keymap names component is specified which does not begin with ‘+’ or ‘|’ and does not contain ‘%’, then ‘%’ refers to the description generated by the keymap names component. Otherwise, it refers to the keyboard description for *device_spec*.
- The ‘+’ operator specifies that the following component should *override* the currently assembled description; any definitions that are present in both components are taken from the second.
- The ‘|’ operator specifies that the next specified component should *augment* the currently assembled description; any definitions that are present in both components are taken from the first.
- If the component expression begins with an operator, a leading ‘%’ is implied.
- If any unknown or illegal characters appear anywhere in the expression, the entire expression is invalid and is ignored.

For example, if *names->symbols* contained the expression “+de”, it specifies that the default member of the “de” class of symbols should be applied to the current keyboard mapping, overriding any existing definitions (It could also be written “+de(default)”).

Here is 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 ASCII keyboard for Acme (*acme(ascii)*), overrides them with definitions for the basic German keyboard (*de(basic)*) and then applies the definitions from the default iso9995-3 keyboard (*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).

Note The interpretation of the above expression components (acme, ascii, de, basic, iso9995-3) is not defined by Xkb; only the operations and their ordering are.

Note that the presence of a keymap *names* component which does not contain ‘%’ (either explicit or implied by virtue of an expression starting with an operator) indicates a description which is independent of the keyboard description for the device specified in *device_spec*. The same is true of requests in which the keymap names component is empty and all five other names components contain expressions void of references to ‘%’. Requests of this form allow you to deal with keyboard definitions independent of any actual device.

The server parses all non-NULL fields in *names* and uses them to build a keyboard description. However, before parsing the expressions in *names*, the server ORs the bits in *want* and *need* together and examines the result in relationship to the expressions in *names*. Table 20.3 identifies the components which are required for each of the possible bits in *want* or *need*. If a required component has not been specified in the *names* structure (the corresponding field is NULL), the server substitutes the expression “%”, resulting in the component values being taken from *device_spec*. In addition, if *load* is True, the server modifies *names* if necessary (again using a “%” entry) to insure all of the following fields are non-NULL: *types*, *keycodes*, *symbols*, and *compat*.

Table 20.3 want and need mask bits and required names components

<i>want or need mask bit</i>	<i>Required names Components</i>	<i>value</i>
XkbGBN_TypesMask	Types	(1L<<0)
XkbGBN_CompatMapMask	Compat	(1L<<1)
XkbGBN_ClientSymbolsMask	Types + Symbols + Keycodes	(1L<<2)
XkbGBN_ServerSymbolsMask	Types + Symbols + Keycodes	(1L<<3)
XkbGBN_SymbolsMask	Symbols	(1L<<1)
XkbGBN_IndicatorMapMask	Compat	(1L<<4)
XkbGBN_KeyNamesMask	Keycodes	(1L<<5)
XkbGBN_GeometryMask	Geometry	(1L<<6)
XkbGBN_OtherNamesMask	Types + Symbols + Keycodes + Compat + Geometry	(1L<<7)
XkbGBN_AllComponentsMask		(0xff)

need specifies a set of keyboard components which the server must be able to resolve in order for *XkbGetKeyboardByName* to succeed; if any of the components specified in *need* cannot be successfully resolved, *XkbGetKeyboardByName* fails.

want specifies a set of keyboard components which the server should attempt to resolve, but which are not mandatory. If the server is unable to resolve any of these components, *XkbGetKeyboardByName* still succeeds. Bits specified in *want* which are also specified in *need* have no effect in the context of *want*.

If *load* is True, the server updates its keyboard description for *device_spec* to match the result of the keyboard description just built. If *load* is False, the server’s description for device *device_spec* is not updated. In all cases, the parts specified by *want* and *need* from the just-built keyboard description are returned.

The *names* structure in an `XkbDescRec` keyboard description record (see Chapter 18) contains one field for each of the five component types used to build a keyboard description. When a keyboard description is built from a set of database components, the corresponding fields in this *names* structure are set to match the expressions used to build the component.

The entire process of building a new keyboard description from the server database of components and returning all or part of it is diagrammed in Figure 20.1:

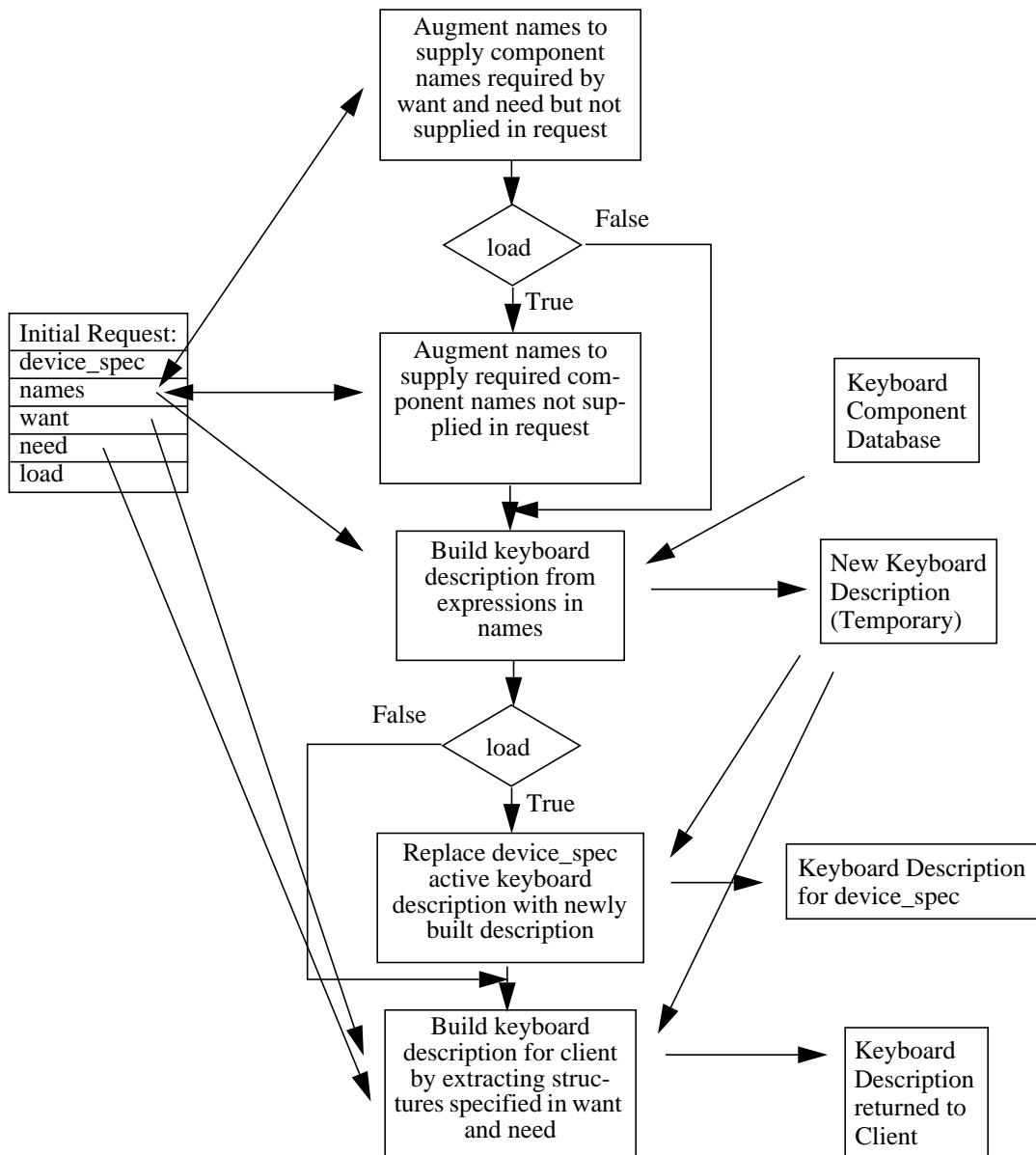


Figure 20.1 Building a new keyboard description from the server database

The information returned to the client in the `XkbDescRec` is essentially the result of a series of calls to extract information from a fictitious device whose description matches

the one just built. The calls corresponding to each of the mask bits are summarized below, together with the `XkbDescRec` components that are filled in.

Table 20.4 XkbDescRec components returned for values of want & need

<i>Request (want+need)</i>	<i>Fills in Xkb components</i>	<i>Equivalent Function Call</i>
<code>XkbGBN_TypesMask</code>	<code>map.types</code>	<code>XkbGetUpdatedMap(dpy, XkbTypesMask, Xkb)</code>
<code>XkbGBN_ServerSymbolsMask</code>	<code>server</code>	<code>XkbGetUpdatedMap(dpy, XkbAllClientInfoMask, Xkb)</code>
<code>XkbGBN_ClientSymbolsMask</code>	<code>map, including map.types</code>	<code>XkbGetUpdatedMap(dpy, XkbAllServerInfoMask, Xkb)</code>
<code>XkbGBN_IndicatorMaps</code>	<code>indicators</code>	<code>XkbGetIndicatorMap(dpy, XkbAllIndicators, Xkb)</code>
<code>XkbGBN_CompatMapMask</code>	<code>compat</code>	<code>XkbGetCompatMap(dpy, XkbAllCompatMask, Xkb)</code>
<code>XkbGBN_GeometryMask</code>	<code>geom</code>	<code>XkbGetGeometry(dpy, Xkb)</code>
<code>XkbGBN_KeyNamesMask</code>	<code>names.keys</code> <code>names.key_aliases</code>	<code>XkbGetNames(dpy, XkbKeyNamesMask XkbKeyAliasesMask, Xkb)</code>
<code>XkbGBN_OtherNamesMask</code>	<code>names.keycodes</code> <code>names.geometry</code> <code>names.symbols</code> <code>names.types</code> <code>map.types[*].lvl_names[*]</code> <code>names.compat</code> <code>names.vmods</code> <code>names.indicators</code> <code>names.groups</code> <code>names.radio_groups</code> <code>names.phys_symbols</code>	<code>XkbGetNames(dpy, XkbAllNamesMask & ~(XkbKeyNamesMask XkbKeyAliasesMask), Xkb)</code>

There is no way to determine which components specified in *want* (but not in *need*) were actually fetched, other than breaking the call into successive calls to `XkbGetKeyboardByName` and specifying individual components.

`XkbGetKeyboardByName` always sets *min_key_code* and *max_key_code* in the returned `XkbDescRec` structure.

`XkbGetKeyboardByName` is synchronous; it sends the request to the server to build a new keyboard description, and waits for the reply. If successful, the return value is non-NULL. `XkbGetKeyboardByName` generates a `BadMatch` protocol error if errors are encountered when building the keyboard description.

If you simply want to obtain information about the current keyboard device, rather than generating a new keyboard description from elements in the server database, call `XkbGetKeyboard` (see section 6.2 on page 29).

```
XkbDescPtr XkbGetKeyboard(dpy, which, device_spec)
    Display *      dpy;          /* connection to X server */
    unsigned int   which;        /* mask of components of XkbDescRec of interest */
    unsigned int   device_spec; /* device id */
```

`XkbGetKeyboard` is used to read the current description for one or more components of a keyboard device. It calls `XkbGetKeyboardByName` as follows:

`XkbGetKeyboardByName(dpy, device_spec, NULL, which, which, False).`

21 Attaching Xkb Actions to X Input Extension Devices

The X input extension allows an X server to support multiple keyboards, as well as other input devices, in addition to the core X keyboard and pointer. The input extension categorizes devices by grouping them into classes. Keyboards, and other input devices with keys, are classified as `KeyClass` devices by the input extension. Other types of devices supported by the input extension include, but are not limited to: mice, tablets, touch-screens, barcode readers, button boxes, trackballs, identifier devices, data gloves, and eye trackers. Xkb provides additional control over all X input extension devices, whether they are `KeyClass` devices or not, as well as the core keyboard and pointer.

If an X server implements support for both the input extension and Xkb, the server implementor determines whether interaction between Xkb and the input extension is allowed. Implementors are free to restrict the effects of Xkb to only the core X keyboard device or allow interaction between Xkb and the input extension.

Several types of interaction between Xkb and the input extension are defined by Xkb. Some or all may be allowed by the X server implementation.

Regardless of whether the server allows interaction between Xkb and the input extension, the following access is provided:

- Xkb functionality for the core X keyboard device and its mapping is accessed via the functions described in the other chapters of this specification.
- Xkb functionality for the core X pointer device is accessed via the *XkbGetDeviceInfo* and *XkbSetDeviceInfo* functions described in this chapter.

If all types of interaction are allowed between Xkb and the input extension, the following additional access is provided:

- If allowed, Xkb functionality for additional `KeyClass` devices supported by the input extension is accessed via those same functions.
- If allowed, Xkb functionality for non-`KeyClass` devices supported by the input extension is also accessed via the *XkbGetDeviceInfo* and *XkbSetDeviceInfo* functions described in this chapter.

Each device has an X Input Extension device id. Each device may have several classes of feedback. For example, there are two types of feedbacks that can generate bells: bell feedback and keyboard feedback (`BellFeedbackClass` and `KbdFeedbackClass`). A device can have more than one feedback of each type; the feedback id identifies the particular feedback within its class.

A keyboard feedback has:

- auto-repeat status (global and per key)
- 32 leds
- a bell

An indicator feedback has:

- up to 32 leds

If the input extension is present and the server allows interaction between the input extension and Xkb, then the core keyboard, the core keyboard indicators, and the core keyboard bells may each be addressed using an appropriate device spec, class, and id. The constant `XkbXIDfltID` may be used as the device id to specify the core keyboard indicators for the core indicator feedback. The particular device id corresponding to the core keyboard

feedback and the core indicator feedback may be obtained by calling *XkbGetDeviceInfo* and specifying *XkbUseCoreKbd* as the *device_spec*; the values will be returned in *dflt_kbd_id* and *dflt_led_id*.

If the server does not allow Xkb access to input extension KeyClass devices, attempts to use Xkb requests with those devices fail with a BadKeyboard error. Attempts to access non-KeyClass input extension devices via *XkbGetDeviceInfo* and *XkbSetDeviceInfo* fail silently if Xkb access to those devices is not supported by the X server.

21.1 XkbDeviceInfoRec

Information about X Input Extension devices is transferred between a client program and the Xkb extension in an *XkbDeviceInfoRec* structure:

```
typedef struct {
    char *name;          /* name for device */
    Atom type;           /* name for class of devices */
    unsigned short device_spec; /* device of interest */
    Bool has_own_state; /* True=>this device has its own state */
    unsigned short supported; /* bits indicating supported capabilities */
    unsigned short unsupported; /* bits indicating unsupported capabilities */
    unsigned short num_btns; /* number of entries in btnActs */
    XkbAction *btnActs; /* button actions */
    unsigned short sz_leds; /* total number of entries in leds vector */
    unsigned short num_leds; /* number of valid entries in leds vector */
    unsigned short dflt_kbd_fb; /* input extension id of default (core kbd) indicator */
    unsigned short dflt_led_fb; /* input extension id of default indicator feedback */
    XkbDeviceLedInfoPtr leds; /* LED descriptions */
} XkbDeviceInfoRec, *XkbDeviceInfoPtr;

typedef struct {
    unsigned short led_class; /* class for this LED device */
    unsigned short led_id; /* id for this LED device */
    unsigned int phys_indicators; /* bits for which leds physically present */
    unsigned int maps_present; /* bits for which leds have maps in maps */
    unsigned int names_present; /* bits for which leds are in names */
    unsigned int state; /* 1 bit => corresponding led is on */
    Atom names[XkbNumIndicators]; /* names for leds */
    XkbIndicatorMapRec maps; /* indicator maps for each led */
} XkbDeviceLedInfoRec, *XkbDeviceLedInfoPtr;
```

The *type* field is a registered symbolic name for a class of devices (for example, “TABLET”). If a device is a keyboard (that is, is a member of KeyClass), it has its own state, and *has_own_state* is True. If *has_own_state* is False, the state of the core keyboard is used. The *supported* and *unsupported* fields are masks where each bit indicates a capability. The meaning of the mask bits are listed in Table 21.1 below, together with the fields in the *XkbDeviceInfoRec* structure which are associated with the capability represented by each bit. The same bits are used to indicate the specific information desired in many of the functions described subsequently in this section.

Table 21.1 XkbDeviceInfoRec Mask Bits

<i>Name</i>	<i>XkbDeviceInfoRec Fields Effected</i>	<i>Value</i>	<i>Capability If Set</i>
XkbXI_KeyboardsMask		(1L << 0)	Clients can use all Xkb requests and events with KeyClass devices supported by the input device extension.
XkbXI_ButtonActionsMask	num_btns btn_acts	(1L <<1)	Clients can assign key actions to buttons on non-KeyClass input extension devices.
XkbXI_IndicatorNamesMask	leds->names	(1L <<2)	Clients can assign names to indicators on non-KeyClass input extension devices.
XkbXI_IndicatorMapsMask	leds->maps	(1L <<3)	Clients can assign indicator maps to indicators on non-KeyClass input extension devices.
XkbXI_IndicatorStateMask	leds->state	(1L <<4)	Clients can request the status of indicators on non-KeyClass input extension devices.
XkbXI_IndicatorsMask	sz_leds num_leds leds->*	(0x1c)	XkbXI_IndicatorNamesMask XkbXI_IndicatorMapsMask XkbXI_IndicatorStateMask
XkbXI_UnsupportedFeaturesMask	unsupported	(1L <<15)	
XkbXI_AllDeviceFeaturesMask	Those selected by Value column masks	(0x1e)	XkbXI_IndicatorsMask XkbSI_ButtonActionsMask
XkbXI_AllFeaturesMask	Those selected by Value column masks	(0x1f)	XkbSI_AllDeviceFeaturesMask XkbSI_KeyboardsMask
XkbXI_AllDetailsMask	Those selected by Value column masks	(0x801f)	XkbXI_AllFeaturesMask XkbXI_UnsupportedFeaturesMask

The *name*, *type*, *has_own_state*, *supported*, and *unsupported* fields are always filled in when a valid reply is returned from the server involving an XkbDeviceInfoRec. All of the other fields are modified only if the particular function asks for them.

21.2 Querying Xkb Features for Non-KeyClass Input Extension Devices

To determine whether the X server allows Xkb access to particular capabilities of input devices other than the core X keyboard, or to determine the status of indicator maps, indicator names or button actions on a non-KeyClass extension device, call *XkbGetDeviceInfo*.

```
XkbDeviceInfoPtr XkbGetDeviceInfo(dpy, which, device_spec, ind_class, ind_id)
    Display *          dpy;           /* connection to X server */
    unsigned int       which;         /* mask indicating information to return */
    unsigned int       device_spec;   /* device id, or XkbUseCoreKbd */
    unsigned int       ind_class;     /* feedback class for indicator requests */
    unsigned int       ind_id;        /* feedback id for indicator requests */
```

XkbGetDeviceInfo returns information about the input device specified by *device_spec*. Unlike the *device_spec* parameter of most Xkb functions, *device_spec* does not need to be

a keyboard device. It must, however, indicate either the core keyboard or a valid X Input Extension device.

The *which* parameter is a mask specifying optional information to be returned. It is an inclusive OR of one or more of the values from Table 21.1, and causes the returned `XkbDeviceInfoRec` to contain values for the corresponding fields specified in the table.

The `XkbDeviceInfoRec` returned by `XkbGetDeviceInfo` always has values for *name* (may be a null string, “”), *type*, *supported*, *unsupported*, *has_own_state*, *dflt_kbd_fd*, and *dflt_kbd_fb*. Other fields are filled in as specified by *which*.

Upon return, the *supported* field will be set to the inclusive OR of zero or more bits from Table 21.1; each bit set indicates an optional Xkb extension device feature supported by the server implementation, and a client may modify the associated behavior.

If the `XkbButtonActionsMask` bit is set in *which*, the `XkbDeviceInfoRec` returned will have the button actions (*btn_acts* field) filled in for all buttons.

If *which* includes one of the bits in `XkbXI_IndicatorsMask`, the feedback class of the indicators must be specified in *ind_class*, and the feedback id of the indicators must be specified in *ind_id*. If the request does not include any of the bits in `XkbXI_IndicatorsMask`, the *ind_class* and *ind_id* parameters are ignored. The class and id can be obtained via the input device extension `XListInputDevices` request.

If any of the `XkbXI_IndicatorsMask` bits are set in *which*, the `XkbDeviceInfoRec` returned will have filled in the portions of the *leds* structure corresponding to the indicator feedback identified by *ind_class* and *ind_id*. The *leds* vector of the `XkbDeviceInfoRec` is allocated if necessary and *sz_leds* and *num_leds* filled in. The *led_class*, *led_id* and *phys_indicators* fields of the *leds* entry corresponding to *ind_class* and *ind_id* are always filled in. If *which* contains `XkbXI_IndicatorNamesMask`, the *names_present* and *names* fields of the *leds* structure corresponding to *ind_class* and *ind_id* are returned. If *which* contains `XkbXI_IndicatorStateMask`, the corresponding *state* field; is updated. If *which* contains `XkbXI_IndicatorMapsMask`, the *maps_present* and *maps* fields are updated.

Xkb provides convenience functions to request subsets of the information available via `XkbGetDeviceInfo`. These convenience functions mirror some of the mask bits. The functions all take an `XkbDeviceInfoPtr` as an input argument, and operate on the X Input Extension device specified by the *device_spec* field of the structure. Only the parts of the structure indicated in the function description are updated. The `XkbDeviceInfoRec` structure used in the function call can be obtained by calling `XkbGetDeviceInfo`, or can be allocated by calling `XkbAllocDeviceInfo` (see section 21.3).

These convenience functions are described below.

To query the button actions associated with an X Input Extension device, use `XkbGetDeviceButtonActions`.

Status	XkbGetDeviceButtonActions (<i>dpy</i> , <i>device_info</i> , <i>all_buttons</i> , <i>first_button</i> , <i>num_buttons</i>)
Display *	<i>dpy</i> ; /* connection to X server */
<code>XkbDeviceInfoPtr</code>	<i>device_info</i> ; /* structure to update with results */
Bool	<i>all_buttons</i> ; /* True => get information for all buttons */
unsigned int	<i>first_button</i> ; /* number of first button for which info desired */
unsigned int	<i>num_buttons</i> ; /* number of buttons for which info is desired */

XkbGetDeviceButtonActions queries the server for the desired button information for the device indicated by the *device_spec* field of *device_info*, and waits for a reply. If successful, *XkbGetDeviceButtonActions* backfills the button actions (*btn_acts* field of *device_info*) for only the requested buttons, updates the *name*, *type*, *supported*, and *unsupported* fields, and returns Success.

all_buttons, *first_button* and *num_buttons* specify the device buttons for which actions should be returned. Setting *all_buttons* to True requests actions for all device buttons; if *all_buttons* is False, *first_button* and *num_buttons* specify a range of buttons for which actions are requested.

If a compatible version of Xkb is not available in the server or the Xkb extension has not been properly initialized, *XkbGetDeviceButtonActions* returns BadAccess. If allocation errors occur, a BadAlloc status is returned. If the specified device (*device_info->device_spec*) is invalid, a BadKeyboard status is returned. If the device has no buttons, a BadMatch status is returned. If *first_button* and *num_buttons* specify illegal buttons, a BadValue status is returned.

To query the indicator names, maps, and state associated with an LED feedback of an input extension device, use *XkbGetDeviceLedInfo*.

```
Status XkbGetDeviceLedInfo(dpy, device_info, led_class, led_id, which)
Display *          dpy;           /* connection to X server */
XkbDeviceInfoPtr  device_info;    /* structure to update with results */
unsigned int       led_class;     /* LED feedback class assigned by input extension */
unsigned int       led_id;        /* LED feedback id assigned by input extension */
unsigned int       which;        /* mask indicating desired information */
```

XkbGetDeviceLedInfo queries the server for the desired led information for the feedback specified by *led_class* and *led_id* for the X input extension device indicated by *device_spec->device_info*, and waits for a reply. If successful, *XkbGetDeviceLedInfo* backfills the relevant fields of *device_info* as determined by *which* with the results and returns Success. Valid values for *which* are the inclusive OR of any of XkbXI_IndicatorNamesMask, XkbXI_IndicatorMapsMask, and XkbXI_IndicatorStateMask.

The fields of *device_info* which are filled in when this request succeeds are *name*, *type*, *supported*, and *unsupported*, and portions of the *leds* structure corresponding to *led_class* and *led_id* as indicated by the bits set in *which*. The *device_info->leds* vector is allocated if necessary and *sz_leds* and *num_leds* filled in. The *led_class*, *led_id* and *phys_indicators* fields of the *device_info->leds* entry corresponding to *led_class* and *led_id* are always filled in.

If *which* contains XkbXI_IndicatorNamesMask, the *names_present* and *names* fields of the *device_info->leds* structure corresponding to *led_class* and *led_id* are updated, if *which* contains XkbXI_IndicatorStateMask, the corresponding *state* field is updated, and if *which* contains XkbXI_IndicatorMapsMask, the *maps_present* and *maps* fields are updated.

If a compatible version of Xkb is not available in the server or the Xkb extension has not been properly initialized, *XkbGetDeviceLedInfo* returns BadAccess. If allocation errors occur, a BadAlloc status is returned. If the device has no indicators, a BadMatch error is returned. If *ledClass* or *ledID* have illegal values, a BadValue error is returned. If they

have legal values but do not specify a feedback that contains LEDs and is associated with the specified device, a BadMatch error is returned.

21.3 Allocating, Initializing, and Freeing the XkbDeviceInfoRec Structure

Obtain an XkbDeviceInfoRec structure by calling *XkbGetDeviceInfo*, or by calling *XkbAllocDeviceInfo*.

```
XkbDeviceInfoPtr XkbAllocDeviceInfo(device_spec, n_buttons, sz_leds)
    unsigned int      device_spec; /* device id with which structure will be used */
    unsigned int      n_buttons;   /* number of button actions to allocate space for */
    unsigned int      sz_leds;    /* number of LED feedbacks to allocate space for */
```

XkbAllocDeviceInfo allocates space for an XkbDeviceInfoRec structure and initializes that structure's *device_spec* field with the device id specified by *device_spec*. If *n_buttons* is non-zero, *n_buttons* XkbActions are linked into the XkbDeviceInfoRec structure and initialized to zero. If *sz_leds* is nonzero, *sz_leds* XkbDeviceLedInfoRec structures are also allocated and linked into the XkbDeviceInfoRec structure. If you request XkbDeviceLedInfoRec structures be allocated using this request, you must initialize them explicitly.

Obtain an XkbDeviceLedInfoRec structure by calling *XkbAllocDeviceLedInfo*.

Status **XkbAllocDeviceLedInfo**(devi, num_needed)

```
XkbDeviceInfoPtr   device_info; /* structure in which to allocate led space */
int               num_needed; /* number of indicators to allocate space for */
```

XkbAllocDeviceLedInfo allocates space for an XkbDeviceLedInfoRec and places it in *device_info*. If *num_needed* is nonzero, *num_needed* XkbIndicatorMapRec structures are also allocated and linked into the XkbDeviceLedInfoRec structure. If you request XkbIndicatorMapRec structures be allocated using this request, you must initialize them explicitly. All other fields are initialized to zero.

To initialize an XkbDeviceLedInfoRec structure, call *XkbAddDeviceLedInfo*.

XkbDeviceLedInfoPtr **XkbAddDeviceLedInfo**(device_info, led_class, led_id)

```
XkbDeviceInfoPtr   device_info; /* structure in which to add led info */
unsigned int       led_class;   /* input extension class for LED device of interest */
unsigned int       led_id;     /* input extension id for LED device of interest */
```

XkbAddDeviceLedInfo first checks to see if an entry matching *led_class* and *led_id* already exists in the *device_info->leds* array. If it finds a matching entry, it returns a pointer to that entry. Otherwise, it checks to be sure there is at least one empty entry in *device_info->leds*, and extends it if there is not enough room. It then increments *device_info->num_leds* and fills in the next available entry in *device_info->leds* with *led_class* and *led_id*.

If successful, *XkbAddDeviceLedInfo* returns a pointer to the XkbDeviceLedInfoRec structure which was initialized. If unable to allocate sufficient storage, or if *device_info* points to an invalid XkbDeviceInfoRec structure, or if *led_class* or *led_id* are inappropriate, *XkbAddDeviceLedInfo* returns NULL.

To allocate additional space for button actions in an `XkbDeviceInfoRec` structure, call `XkbResizeDeviceButtonActions`.

Status **`XkbResizeDeviceButtonActions`**(*device_info*, *new_total*)

<code>XkbDeviceInfoPtr</code>	<i>device_info</i> ; /* structure in which to allocate button actions */
<code>unsigned int</code>	<i>new_total</i> ; /* new total number of button actions needed */

`XkbResizeDeviceButton` reallocates space, if necessary, to make sure there is room for a total of *new_total* button actions in the *device_info* structure. Any new entries allocated are zeroed. If successful, `XkbResizeDeviceButton` returns `Success`. If *new_total* is zero, all button actions are deleted, *device_info->num_btns* is set to zero, and *device_info->btn_acts* is set to `NULL`. If *device_info* is invalid or *new_total* is greater than 255, `BadValue` is returned. If a memory allocation failure occurs, a `BadAlloc` is returned.

To free an `XkbDeviceInfoRec` structure, call `XkbFreeDeviceInfo`.

`void XkbFreeDeviceInfo`(*device_info*, *which*, *free_all*)

<code>XkbDeviceInfoPtr</code>	<i>device_info</i> ; /* pointer to <code>XkbDeviceInfoRec</code> in which to free items */
<code>unsigned int</code>	<i>which</i> ; /* mask of components of <i>device_info</i> to free */
<code>Bool</code>	<i>free_all</i> ; /* True => free everything, including <i>device_info</i> */

If *free_all* is `True`, the `XkbFreeDeviceInfo` frees all components of *device_info* and the `XkbDeviceInfoRec` structure pointed to by *device_info* itself. If *free_all* is `False`, the value of *which* determines which subcomponents are freed. *which* is an inclusive OR of one or more of the values from Table 21.1 on page 200. If *which* contains `XkbXI_ButtonActionsMask`, all button actions associated with *device_info* are freed, *device_info->btn_acts* is set to `NULL`, and *device_info->num_btns* is set to zero. If *which* contains all bits in `XkbXI_IndicatorsMask`, all `XkbDeviceLedInfoRec` structures associated with *device_info* are freed, *device_info->leds* is set to `NULL`, and *device_info->sz_leds* and *device_info->num_leds* are set to zero. If *which* contains `XkbXI_IndicatorMapsMask`, all indicator maps associated with *device_info* are cleared, but the number of leds and the leds structures themselves are preserved. If *which* contains `XkbXI_IndicatorNamesMask`, all indicator names associated with *device_info* are cleared but the number of leds and the leds structures themselves are preserved. If *which* contains `XkbXI_IndicatorStateMask`, the indicator state associated with the *device_info* leds are set to zeros but the number of leds and the leds structures themselves are preserved.

21.4 Setting Xkb Features for Non-KeyClass Input Extension Devices

The Xkb extension allows clients to assign any key action to either 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.

Key actions assigned to core X pointer buttons or input extension device buttons cause key events to be generated as if they had originated from the core X keyboard.

Xkb implementations are required to support key 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_ButtonActionsMask` bit in the *supported* field of an `XkbDeviceInfoRec` structure.

If a client attempts to modify valid characteristics of a device using an implementation which does not support modification of those characteristics, no protocol error is generated. Instead, the server reports a failure for the request; it also sends an `XkbExtensionDeviceNotify` event to the client which issued the request if the client has selected to receive these events.

To change characteristics of an X Input Extension device in the server, first modify a local copy of the device structure and then use either `XkbSetDeviceInfo` or, to save network traffic, use an `XkbDeviceChangesRec` structure (see section 21.6) and call `XkbChangeDeviceInfo` to download the changes to the server.

To modify some or all of the characteristics of an X Input Extension device, call `XkbSetDeviceInfo`.

```
Bool XkbSetDeviceInfo(dpy, which, device_info)
    Display *          dpy;           /* connection to X server */
    unsigned int       which;         /* mask indicating characteristics to modify */
    XkbDeviceInfoPtr  device_info;   /* structure defining the device and modifications */
```

`XkbSetDeviceInfo` sends a request to the server to modify the characteristics of the device specified in the *device_info* structure. The particular characteristics modified are identified by the bits set in *which*, and take their values from the relevant fields in *device_info* (see Table 21.1 on page 200). `XkbSetDeviceInfo` returns True if the request was successfully sent to the server. If the X server implementation does not allow interaction between the X input extension and the Xkb Extension, the function does nothing and returns False.

The *which* parameter specifies which aspects of the device should be changed, and is a bit-mask composed of an inclusive OR or one or more of the following bits:

`XkbXI_ButtonActionsMask`, `XkbXI_IndicatorNamesMask`,
`XkbXI_IndicatorMapsMask`. If the features requested to be manipulated in *which* are valid for the device, but the server does not support assignment of one or more of them, that particular portion of the request is ignored.

If the device specified in *device_info->device_spec* does not contain buttons and a request affecting buttons is made, or the device does not contain indicators and a request affecting indicators is made, a `BadMatch` protocol error results.

If the `XkbXI_ButtonActionsMask` bit is set in the *supported* mask returned by `XkbGetDeviceInfo`, the Xkb extension allows applications to assign key actions to buttons on input extension devices other than the core keyboard device. If the `XkbXI_ButtonActionsMask` is set in *which*, the actions for all buttons specified in *device_info* are set to the `XkbActions` specified in *device_info->btn_acts*. If the number of buttons requested to be updated is not valid for the device, `XkbSetDeviceInfo` returns False and a `BadValue` protocol error results.

If the `XkbXI_IndicatorMaps` and / or `XkbXI_IndicatorNamesMask` bit is set in the *supported* mask returned by `XkbGetDeviceInfo`, the Xkb extension allows applications to assign maps and / or names to the indicators of non-keyboard extension devices. If supported, maps and / or names can be assigned to all extension device indicators, whether they are part of a keyboard feedback or part of an indicator feedback.

If the `XkbXI_IndicatorMapsMask` and / or `XkbXI_IndicatorNamesMask` flag is set in *which*, the indicator maps and / or names for all *device_info->num_leds* indicator devices specified in *device_info->leds* are set to the maps and / or names specified in

device_info->leds. *device_info->leds->led_class* and *led_id* specify the input extension class and device id for each indicator device to modify; if they have invalid values, a BadValue protocol error results and *XkbSetDeviceInfo* returns False. If they have legal values but do not specify a keyboard or indicator class feedback for the device in question, a BadMatch error results. If any of the values in *device_info->leds->names* are not a valid Atom or None, a BadAtom protocol error results.

Xkb provides convenience functions to modify subsets of the information accessible via *XkbSetDeviceInfo*. Only the parts of the structure indicated in the function description are modified. These convenience functions are described below.

To change only the button actions for an input extension device, call *XkbSetDeviceButtonActions*.

```
Bool XkbSetDeviceButtonActions(dpy, device, first_button, num_buttons, actions)
    Display *          dpy;           /* connection to X server */
    XkbDeviceInfoPtr device_info;   /* structure defining the device and modifications */
    unsigned int       first_button;  /* number of first button to update, 0 relative */
    unsigned int       num_buttons;  /* number of buttons to update */
```

XkbSetDeviceButtonActions assigns actions to the buttons of the device specified in *device_info->device_spec*. Actions are assigned to *num_buttons* buttons beginning with *first_button*, and are taken from the actions specified in *device_info->btn_acts*.

If the server does not support assignment of Xkb actions to extension device buttons, *XkbSetDeviceButtonActions* has no effect and returns False. If the device has no buttons or if *first_button* or *num_buttons* specify buttons outside of the valid range as determined by *device_info->num_btns*, the function has no effect and returns False. Otherwise, *XkbSetDeviceButtonActions* sends a request to the server to change the actions for the specified buttons and returns True.

If the actual request sent to the server involved illegal button numbers, a BadValue protocol error is generated. If an invalid device identifier is specified in *device_info->device_spec*, a BadKeyboard protocol error results. If the actual device specified in *device_info->device_spec* does not contain buttons and a request affecting buttons is made, a BadMatch protocol error is generated.

21.5 XkbExtensionDeviceNotify Event

The Xkb extension generates *XkbExtensionDeviceNotify* events when the status of an input extension device changes, or when an attempt is made to use an Xkb feature which is not supported by a particular device.

Note Events indicating an attempt to use an unsupported feature are delivered only to the client requesting the event.

To track changes to the status of input extension devices or attempts to use unsupported features of a device, select to receive *XkbExtensionDeviceNotify* events by calling either *XkbSelectEvents* or *XkbSelectEventDetails* (see section 4.3).

To receive *XkbExtensionDeviceNotify* events under all possible conditions, call *XkbSelectEvents* and pass *XkbExtensionDeviceNotifyMask* in both *bits_to_change* and *values_for_bits*.

The `XkbExtensionDeviceNotify` event has no event details. However, you can call `XkbSelectEventDetails` using `XkbExtensionDeviceNotify` as the `event_type` and specifying `XkbAllExtensionDeviceMask` in `bits_to_change` and `values_for_bits`. This has the same effect as a call to `XkbSelectEvents`.

The structure for `XkbExtensionDeviceNotify` events is:

```
typedef struct {
    int             type;           /* Xkb extension base event code */
    unsigned long   serial;         /* X server serial number for event */
    Bool            send_event;     /* True => synthetically generated */
    Display *       display;        /* server connection where event generated */
    Time            time;          /* server time when event generated */
    int             xkb_type;       /* XkbExtensionDeviceNotifyEvent */
    int             device;         /* Xkb device id, will not be XkbUseCoreKbd */
    unsigned int    reason;        /* reason for the event */
    unsigned int    supported;     /* mask of supported features */
    unsigned int    unsupported;   /* unsupported features this client attempted to use */
    int             first_btn;     /* first button that changed */
    int             num_btns;      /* number of buttons that changed */
    unsigned int    leds_defined;  /* indicators with names or maps */
    unsigned int    led_state;     /* current state of the indicators */
    int             led_class;     /* feedback class for LED changes */
    int             led_id;         /* feedback id for LED changes */
} XkbExtensionDeviceNotifyEvent;
```

The `XkbExtensionDeviceNotify` event has fields enabling it to report changes in the state (on/off) of all of the buttons for a device, but only for one LED feedback associated with a device. You will get multiple events when more than one led feedback changes state or configuration.

21.6 Tracking Changes to Extension Devices

Changes to an Xkb extension device may be tracked by listening to `XkbDeviceExtensionNotify` events and accumulating the changes in an `XkbDeviceChangesRec` structure. The changes noted in the structure may then be used in subsequent operations to update either a server configuration or a local copy of an Xkb extension device configuration. The changes structure is defined below:

```
typedef struct _XkbDeviceChanges {
    unsigned int    changed;        /* bits indicating what has changed */
    unsigned short  first_btn;      /* number of first button which changed, if any */
    unsigned short  num_btns;       /* number of buttons which have changed */
    XkbDeviceLedChangesRec leds;
} XkbDeviceChangesRec,*XkbDeviceChangesPtr;

typedef struct _XkbDeviceLedChanges {
    unsigned short  led_class;     /* class of this indicator feedback bundle */
    unsigned short  led_id;         /* id of this indicator feedback bundle */
    unsigned int    names;          /* bits indicating which names have changed */
    unsigned int    maps;           /* bits indicating which maps have changed */
    struct _XkbDeviceLedChanges *next; /* link to indicator change record for next set */
} XkbDeviceLedChangesRec,*XkbDeviceLedChangesPtr;
```

A local description of the configuration and state of a device may be kept in an `XkbDeviceInfoRec` structure. The actual state or configuration of the device may change because of `XkbSetDeviceInfo` and `XkbSetButtonActions` requests made by clients, or by user interaction with the device. The X server sends an `XkbExtensionDeviceNotify` event to all interested clients when the state of any buttons or indicators or the configuration of the buttons or indicators on the core keyboard or any input extension device changes. The event reports the state of indicators for a single indicator feedback, and the state of up to 128 buttons. If more than 128 buttons or more than one indicator feedback are changed, the additional buttons and indicator feedbacks are reported in subsequent events. Xkb provides functions with which you can track changes to input extension devices by noting the changes which were made, and then requesting the changed information from the server.

To note device changes reported in an `XkbExtensionDeviceNotify` event, call `XkbNoteDeviceChanges`.

```
void XkbNoteDeviceChanges (old, new, wanted)
    XkbDeviceChangesPtr      old;          /* structure tracking state changes */
    XkbExtensionDeviceNotifyEvent * new;        /* event indicating state changes */
    unsigned int              wanted;       /* mask indicating changes to note
                                                */

```

The *wanted* field specifies the changes which should be noted in *old*, and is composed of the bitwise inclusive OR of one or more of the masks from Table 21.1 on page 200. The *reason* field of the event in *new* indicates the types of changes the event is reporting. `XkbNoteDeviceChanges` updates the `XkbDeviceChangesRec` specified by *old* with the changes that are both specified in *wanted* and contained in *new->reason*.

To update a local copy of the state and configuration of an X input extension device with the changes previously noted in an `XkbDeviceChangesRec` structure, call `XkbGetDeviceInfoChanges`.

To query the changes that have occurred in the button actions or indicator names and indicator maps associated with an input extension device, use `XkbGetDeviceInfoChanges`.

```
Status XkbGetDeviceInfoChanges(dpy, device_info, changes)
    Display *           dpy;          /* connection to X server */
    XkbDeviceInfoPtr   device_info;  /* structure to update with results */
    XkbDeviceChangesPtr changes;     /* contains notes of changes which have occurred */

```

The *changes->changed* field indicates which attributes of the device specified in *changes->device* have changed. The parameters describing the changes are contained in the other fields of *changes*. `XkbGetDeviceInfoChanges` uses that information to call `XkbGetDeviceInfo` to obtain the current status of those attributes which have changed. It then updates the local description of the device in *device_info* with the new information.

To update the server's description of a device with the changes noted in an `XkbDeviceChangesRec`, call `XkbChangeDeviceInfo`.

```
Bool XkbChangeDeviceInfo (dpy, device_info, changes)
    Display *           dpy;          /* connection to X server */
    XkbDeviceInfoPtr   device_info;  /* local copy of device state and configuration */
    XkbDeviceChangesPtr changes;     /* note specifying changes in device_info */

```

XkbChangeDeviceInfo updates the server's description of the device specified in *device_info->device_spec* with the changes specified in *changes* and contained in *device_info*. The update is made by an *XkbSetDeviceInfo* request.

22 Debugging Aids

The debugging aids are intended for use primarily by Xkb implementors, and are optional in any implementation.

There are two bitmasks which may be used to control debugging. One bitmask controls the output of debugging information, and the other controls behavior. Both bitmasks are initially all zeros.

To change the values of any of the debug controls, call *XkbSetDebuggingFlags*.

```
Bool XkbSetDebuggingFlags(display, mask, flags, msg, ctrls_mask, ctrls, ret_flags, ret_ctrls)
    Display *          display;      /* connection to X server */
    unsigned int       mask;        /* mask selecting debug output flags to change */
    unsigned int       flags;       /* values for debug output flags selected by mask */
    char *            msg;         /* message to print right now */
    unsigned int       ctrls_mask;   /* mask selecting debug controls to change */
    unsigned int       ctrls;       /* values for debug controls selected by ctrls_mask */
    */
    unsigned int *     ret_flags;    /* resulting state of all debug output flags */
    unsigned int *     ret_ctrls;   /* resulting state of all debug controls */
```

XkbSetDebuggingFlags modifies the debug output flags as specified by *mask* and *flags*, modifies the debug controls flags as specified by *ctrls_mask* and *ctrls*, prints the message *msg*, and backfills *ret_flags* and *ret_ctrls* with the resulting debug output and debug controls flags.

When bits are set in the debug output masks, *mask* and *flags*, Xkb prints debug information corresponding to each bit at appropriate points during its processing. The device to which the output is written is implementation dependent, but is normally the same device to which X server error messages are directed; thus the bits that can be set in *mask* and *flags* is implementation specific. To turn on a debug output selection, set the bit for the output in the *mask* parameter and set the corresponding bit in the *flags* parameter. To turn off event selection for an event, set the bit for the output in the *mask* parameter and do not set the corresponding bit in the *flags* parameter.

When bits are set in the debug controls masks, *ctrls_mask* and *ctrls*, Xkb modifies its behavior according to each controls bit. *ctrls_mask* and *ctrls* are related in the same way that *mask* and *flags* are. The valid controls bits are defined in Table 22.1.

Table 22.1 Debug Control Masks

<i>Debug Control Mask</i>	<i>Value</i>	<i>Meaning</i>
XkbDF_DisableLocks	$(1 << 0)$	Disable actions which lock modifiers

XkbSetDebuggingFlags returns True if successful, and False otherwise. The only protocol error it may generate is BadAlloc, if for some reason it is unable to allocate storage.

XkbSetDebuggingFlags is intended for developer use and may be disabled in production X servers. If it is disabled, *XkbSetDebuggingFlags* has no effect and does not generate any protocol errors.

The message in *msg* is written immediately. The device to which it is written is implementation dependent, but is normally the same device where X server error messages are directed.

Glossary

Allocator

Xkb provides functions, known as allocators, to create and initialize Xkb data structures.

Audible Bell

An audible bell is the sound generated by whatever bell is associated with the keyboard or input extension device, as opposed to any other audible sound generated elsewhere in the system.

Autoreset Controls

The autoreset controls configure the boolean controls to automatically be enabled or disabled at the time a program exits.

Base Group

The group in effect as a result of all actions other than a previous lock or latch request; the base group is transient. For example, the user pressing and holding a group shift key which shifts to Group2 would result in the base group being group 2 at that point in time. Initially, base group is always Group1.

Base Modifiers

Modifiers which are turned on as a result of some actions other than previous lock or latch requests; base modifiers are transient. For example, the user pressing and holding a key bound to the Shift modifier would result in Shift being a base modifier at that point in time.

Base Event Code

A number assigned by the X server at run time which is assigned to the extension to identify events from that extension.

Base State

The base group and base modifiers represent keys that are physically or logically down; these constitute the base state.

Boolean Controls

Global keyboard controls which may be selectively enabled and disabled under program control, and which may be automatically set to an on or off condition upon client program exit.

Canonical Key Types

The canonical key types are predefined key types which describe the types of keys available on most keyboards. The definitions for the canonical key types are held in the first `XkbNumRequiredTypes` entries of the *types* field of the client map, and are indexed using the following constants:

`XkbOneLevelIndex`
`XkbTwoLevelIndex`
`XkbAlphabeticIndex`
`XkbKeypadIndex`

Client Map

The key mapping information needed to convert arbitrary keycodes to symbols.

Compat Name

The *compat* name is a string which provides some information about the rules used to bind actions to keys that are changed using core protocol requests.

Compatibility State

When an Xkb-extended X server connects to an Xkb-unaware client, the compatibility state remaps the keyboard group into a core modifier whenever possible.

Compatibility Grab State

The grab state that results from applying the compatibility map to the Xkb grab state.

Compatibility Map

The definition of how to map core protocol keyboard state to Xkb keyboard state.

Component Expression

An expression used to describe server keyboard database components to be loaded. It describes the order in which the components should be loaded, and the rules by which duplicate attributes should be resolved.

Compose Processing

The process of mapping a series of keysyms to a string is known as compose processing.

Consumed Modifier

Xkb normally consumes modifiers in determining the appropriate symbol for an event, that is, the modifiers are not considered during any of the later stages of event processing. For those rare occasions when a modifier should be considered despite having been used to look up a symbol, key types include an optional *preserve* field.

Core Event

An event created from the core X server.

Detectable Auto-repeat

Detectable auto-repeat allows a client to detect an auto-repeating key. If a client requests and the server supports detectable auto-repeat, Xkb generates KeyRelease events only when the key is physically released. Thus the client receives a number of KeyPress events for that key without intervening KeyRelease events, until the key is finally released, when a KeyRelease event is received.

Effective Group

The effective group is the arithmetic sum of the locked, latched, and base groups. The effective keyboard group is always brought back into range depending on the value of the GroupsWrap control for the keyboard. If an event occurs with an effective group that is legal for the keyboard as a whole, but not for the key in question, the group for that event only is normalized using the algorithm specified by the *group_info* member of the key symbol map (XkbSymMapRec).

Effective Mask

An Xkb modifier definition consists of a set of bit masks corresponding to the eight real modifiers; a similar set of bitmasks corresponding to the 16 named virtual modifiers; and an effective mask. The effective mask represents the set of all real modifiers which can logically be set either by setting any of the real modifiers or by setting any of the virtual modifiers in the definition.

Effective Modifier

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

Extension Device

Any keyboard or other input device recognized by the X input extension.

Global Keyboard Controls

Controls that affect the way Xkb generates key events. The controls affect all keys, as opposed to per-key controls that are for a single key. Global controls include

- RepeatKeys Control
- DetectableAuto-repeat
- SlowKeys
- BounceKeys
- StickyKeys
- MouseKeys
- MouseKeysAccel
- AccessXKeys
- AccessXTIMEout
- AccessXFeedback
- Overlay1
- Overlay2
- EnabledControls

Grab State

The grab state is the state used when matching events to passive grabs. It consists of the grab group and the grab modifiers.

Group

See Keysym Group

Group Index

A number used as the internal representation for a group number. Group1 through Group4 have indices of 0 through 3.

Groups Wrap Control

If a group index 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. GroupsWrap can have the following values:

- WrapIntoRange
- ClampIntoRange
- RedirectIntoRange

Key Type

An attribute of a key which identifies which modifiers affect the shift level of a key, and the number of groups on the key.

Key Width

The maximum number of shift levels in any group for the key type associated with a key.

Keysym Group

A keysym group is a logical state of the 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, Group1 could be the English alphabet, and Group2 could be Greek. Xkb supports up to four different groups for an input device or keyboard. Groups are in the range 1-4 (Group1 - Group4), and are often referred to as G1 - G4 and indexed as 0 - 3.

Indicator

An indicator is a feedback mechanism such as an LED on an input device. Using Xkb, a client application 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.

Indicator Feedback

An indicator feedback describes the state of a bank of up to 32 lights. It has a mask where each bit corresponds to a light, and an associated value mask that specifies which lights are on or off.

Indicator Map

An indicator has its own set of attributes that specify if clients can explicitly set its state and if it tracks the keyboard state. The indicator map is the collection of these attributes for each indicator and is held in the *maps* array, which is an array of `XkbIndicatorRec` structures.

Input Extension

An extension to the core X protocol which allows an X server to support multiple keyboards, as well as other input devices, in addition to the core X keyboard and pointer. Other types of devices supported by the input extension include, but are not limited to: mice, tablets, touchscreens, barcode readers, button boxes, trackballs, identifier devices, data gloves, and eye trackers.

Key Action

A key action consists of an operator and some optional data. 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. 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

Key Alias

A key alias is a symbolic name for a specific physical key. Key aliases allow the keyboard layout designer to assign multiple key names to a single key. This allows the keyboard layout designer to refer to keys using either their position or their “function.” Key aliases can be specified both in the symbolic names component and in the 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, you should consider the definitions from the geometry before considering the definitions from the symbolic names section.

Key Behavior

The *behaviors* field of the server map is an array of `XkbBehavior`, indexed by keycode, and contains the behavior for each key. 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 include:

- `XkbKB_Default`
- `XkbKB_Lock`
- `XkbKB_RadioGroup`
- `XkbKB_Overlay1`
- `XkbKB_Overlay2`

Key Symbol Map

A key symbol map describes the symbols bound to a key and the rules to be used to interpret those symbols. It is an array of `XkbSymMapRec` structures indexed by keycode.

Key Type

Key types are used to determine the shift level of a key given the current state of the keyboard. There is one key type for each group for a key. Key types are defined using the `XkbKeyTypeRec` and `XkbKTMMapEntryRec` structures. Xkb allows up to `XkbMaxKeyTypes` (255) key types to be defined, but requires at least `XkbNumRequiredTypes` (4) predefined types to be in a key map.

Keyboard Bells

The sound the default bell makes when rung is the system bell or the default keyboard bell. Some input devices may have more than one bell, identified by *bell_class* and *bell_id*.

Keyboard Components

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

Keyboard Feedback

A keyboard feedback includes the following:

- keyclick volume
- bell volume
- bell pitch
- bell duration
- global auto-repeat
- per key auto-repeat
- 32 leds

Key Width, Key Type Width

The maximum number of shift levels for a type is referred to as the width of a key type.

Keyboard Geometry

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

Keyboard Geometry Name

The keyboard geometry name describes the physical location, size and shape of the various keys on the keyboard and is part of the `XkbNamesRec` structure.

Keyboard State

Keyboard state encompasses all of the transitory information necessary to map a physical key press or release to an appropriate event.

Keycode

A numeric value returned to the X server when a key on a keyboard is pressed or released, indicating which key is being modulated. Keycode numbers are in the range $1 \leq \text{keycode} \leq \text{max}$, where max is the number of physical keys on the device.

Keycode Name

The keycode name describes the range and meaning of the keycodes returned by the keyboard, and is part of the `XkbNamesRec` structure.

Latched Group

A latched group is a group index that is combined with the base and locked group to form the effective group. It applies only to the next key event that does not change the keyboard state. The latched group can be changed by keyboard activity or via Xkb extension library functions.

Latched Modifier

Latched modifiers are the set of modifiers which are combined with the base modifiers and the locked modifiers to form the effective modifiers. It applies only to the next key event that does not change the keyboard state.

LED

A light emitting diode. However, for the purposes of the X keyboard extension specification, a LED is any form of visual two state indicator which is either on or off.

Locked Group

A locked group is a group index that is combined with the base and latched group to form the effective group. When a group is locked, it supersedes any previous locked group and remains the locked group for all future key events, until a new group is locked. The locked group can be changed by keyboard activity or via Xkb extension library functions.

Locked Modifiers

Locked modifiers are the set of modifiers which are combined with the base modifiers and the latched modifiers to form the effective modifiers. A locked modifier applies to all future key events until it is explicitly unlocked.

Lookup State

The lookup state is composed of the lookup group and the lookup modifiers, and it is the state an Xkb-capable or Xkb-aware client should use to map a keycode to a keysym.

Modifier

A modifier is a logical condition which is either set or unset. The modifiers control the Shift Level selected when a key event occurs. Xkb supports the core protocol eight modifiers (Shift, Lock, Control, and Mod1 through Mod5), called the *real* modifiers. In addition, Xkb extends modifier flexibility by providing a set of sixteen named virtual modifiers, each of which can be bound to any set of the eight real modifiers.

Modifier Key

A modifier key is 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 modifier key may be, for example, a shift key or a control key.

Modifier Definition

An Xkb modifier definition, held in an XkbModsRec, consists of a set of real modifiers, a set of virtual modifiers, and an effective mask. The mask is the union of the real modifiers and the set of real modifiers to which the virtual modifiers map; the mask cannot be explicitly changed.

Non-keyboard Extension Device

An input extension device which is not a keyboard. Other types of devices supported by the input extension include, but are not limited to: mice, tablets, touchscreens, barcode readers, button boxes, trackballs, identifier devices, data gloves, and eye trackers.

Outlines

An outline is a list of one or more points which describes a single closed-polygon, used in the geometry specification for a keyboard.

Physical Indicator Mask

The physical indicator mask is a field in the XkbIndicatorRec which indicates which indicators are bound to physical LEDs on the keyboard; if a bit is set in *phys_indicators*, then the associated indicator has a physical LED associated with it. This field is necessary because some indicators may not have corresponding physical LEDs on the keyboard.

Physical Symbol Keyboard Name

The *symbols* keyboard name identifies the symbols logically 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 *phys_symbols* keyboard name, on the other hand, identifies the symbols actually engraved on the keyboard.

Preserved Modifier

Xkb normally consumes modifiers in determining the appropriate symbol for an event, that is, the modifiers are not considered during any of the later stages of event processing. For those rare occasions when a modifier should be considered despite having been used to look up a symbol, key types include an optional *preserve* field. If a modifier is present in the *preserve* list, it is a preserved modifier.

Radio Group

A radio group is a set of keys whose behavior simulates a set of radio buttons. Once a key in a radio group is pressed, it stays logically depressed until another key in the group is pressed, at which point the previously depressed key is logically released. Consequently, at most one key in a radio group can be logically depressed at one time.

Real Modifier

Xkb supports the eight core protocol modifiers (Shift, Lock, Control, and Mod1 through Mod5); these are called the *real* modifiers, as opposed to the set of sixteen named virtual modifiers which can be bound to any set of the eight real modifiers.

Server Internal Modifiers

Modifiers which the server uses to determine the appropriate symbol for an event; internal modifiers are normally consumed by the server.

Shift Level

One of several states (normally 2 or 3) governing which graphic character is produced when a key is actuated.

Symbol Keyboard Name

The *symbols* keyboard name identifies the symbols logically 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 *phys_symbols* keyboard name, on the other hand, identifies the symbols actually engraved on the keyboard.

Symbolic Name

Xkb supports symbolic names for most components of the keyboard extension. Most of these symbolic names are grouped into the *names* component of the keyboard description.

State Field

The portion of a client-side core protocol event which holds the modifier, group, and button state information pertaining to the event.

Types Name

The *types* name provides some information about the set of key types that can be associated with the keyboard. In addition, each key type can have a name, and each shift level of a type can have a name.

Valuator

A valuator reports a range of values for some entity, like a mouse axis, a slider or a dial.

Virtual Modifier

Xkb provides a set of sixteen named virtual modifiers which can be bound to any set of the eight real modifiers. Each virtual modifier can be bound to any set of the real modifiers (Shift, Lock, Control and Mod1-Mod5).

Virtual Modifier Mapping

Xkb maintains a virtual modifier mapping, which lists the virtual modifiers associated with each key.

Xkb-aware Client

A client application which initializes Xkb extension, and is consequently bound to an Xlib which includes the Xkb extension.

Xkb-capable Client

A client application which makes no Xkb extension Xlib calls, but which is bound to an Xlib which includes the Xkb extension.

Xkb-unaware Client

A client application which makes no Xkb extension Xlib calls, and which is bound to an Xlib which does not include the Xkb extension.

A

AccessX enable/disable bits, table 75
 AccessXFeedback 53, 63, 72
 AccessXKeys 53, 72
 AccessXNotify 64, 65
 AccessXTIMEout 53, 62, 72
 Action modifiers 143
 Actions 140
 changing number of actions bound to key 159
 controls action types, table 153
 detecting key action messages 154
 device button action types 157
 for changing active screen 152
 for changing button number simulated by mouse keys 148
 for changing current group state 144
 for changing state of boolean controls 153
 for changing the state of modifiers 142
 for generating a different keycode for key 155
 for generating DeviceButtonPress and DeviceButtonRelease 157
 for generating messages 154
 for locking modifiers and group 149
 for moving the pointer 146
 for simulating events from device valuators 158
 for simulating pointer button press and release 147
 group action flags, table 145
 group action types, table 145
 ISO action flags, table 150
 message action flags, table 154
 modifier action flags, table 144
 modifier action types, table 143
 obtaining actions for keys from server 159
 pointer action types, table 146
 pointer button action flags, table 148
 pointer button action types, table 148
 switch screen action flags, table 152
 Allocator, glossary entry 211
 AlwaysConsumeShiftAndLock 82
 Audible Bell, glossary entry 211
 AudibleBell 48, 53, 56, 72
 AutoReset 53, 54, 72
 Autoreset Controls, glossary entry 211
 AX_BounceKeyReject 48
 AX_FeatureChange 48
 AX_FeatureOff 48
 AX_FeatureOn 48
 AX_IndicatorChange 48
 AX_IndicatorOff 48
 AX_IndicatorOn 48
 ax_options 75
 ax_options values 63
 AX_SlowKeyAccept 48
 AX_SlowKeyPress 48
 AX_SlowKeyReject 48
 AX_SlowKeyRelease 48
 AX_SlowKeysWarning 48
 AX_StickyLatch 48
 AX_StickyLock 48
 AX_StickyUnlock 48

B

BadAccess 9
 BadAlloc 9
 BadAtom 9
 BadClass 9

BadDevice 9

BadId 9
 BadImplementation 9
 BadKeyboard 4, 9
 BadMatch 9
 BadValue 9
 Base error code 7
 Base event code 7, 14
 Base Event Code, glossary entry 211
 Base group 20, 211
 Base Group, glossary entry 211
 Base modifiers 20, 211
 Base Modifiers, glossary entry 211
 Base State, glossary entry 211
 BeepOnComposeFail 83
 Behavior
 key behaviors, table 160
 keys 160
 obtaining key behaviors from the server 161
 BellFeedbackClass 48
 Bells 47
 audible 48
 BeepOnComposeFail 83
 bell_class and bell_id 48
 detecting 51
 fixed pitch bell only 64
 forcing a server-generated bell 51
 generating bell events 49
 generating named bell events 50
 high and low pitched beeps, rising and falling tones 63
 names 47
 predefined 48
 sounding 49

Boolean controls 53
 actions for changing the state of 153
 Boolean Controls, glossary entry 211

BounceKeys 53, 66, 72
 debounce_delay 75
 delay 66

Bounds
 computing bounding box of a row 105
 computing bounding box of a section 105
 computing bounding box of a shape 104
 keyboard geometry 92
 sections 94
 shapes 93

Buttons, pointer 20

C

Canonical key types 128
 initializing 130
 used in compatibility map 175
 Canonical Key Types, glossary entry 211

Changes data structures 12

ClampInRange 69, 74, 134

Client map 2, 115, 125
 allocating and freeing 122
 key symbol map 132

Client Map, glossary entry 211

Client types
 Xkb-aware 3, 21, 166
 Xkb-capable 3, 21, 166
 Xkb-unaware 3, 21, 166

Colors
 keyboard, key label 92

- listed in geometry description 92
Compat Name, glossary entry 212
Compatibility 3
 allocating and freeing maps 178
 changing the server's map 176
 core keyboard mapping to Xkb keyboard mapping transformation 169
 data structure 168
 data structures, diagram 168
 determining library 6
 diagram 167
 getting map components from server 173
 group maps 168
 map 166
 setting explicit component controls 169, 170, 175
 states 22
 symbol interpretation match criteria, table 171
 symbol interpretations 171
 tracking changes to the map 177
 types of transformations 167
 using the compatibility map 174
 with the core protocol 4
 Xkb keyboard mapping to core keyboard mapping transformations 172
 Xkb state to core protocol state transformation 168
Compatibility Grab State, glossary entry 212
Compatibility Map, glossary entry 212
Compatibility State, glossary entry 212
Component Expression, glossary entry 212
Components, explicit 162
Compose processing controls 82
Compose Processing, glossary entry 212
ComposeLED 83
Composing
 BeepOnComposeFail 83
 ComposeLED 83
 ConsumeKeysOnComposeFail 82
Consumed Modifier, glossary entry 212
ConsumeKeysOnComposeFail 82
ConsumeLookupMods 81
Controls 2
 AccessXFeedback 63
 AccessXTimeout 62
 actions for changing the state of 153
 affecting compose processing 82
 affecting keycode to string translation 81
 allocating and freeing data structure 80
 AlwaysConsumeShiftAndLock 82
 AudibleBell 56
 AutoReset 54
 BeepOnComposeFail 83
 bell behavior 56
 boolean 53, 81
 BounceKeys 66
 changing 77
 changing the state of library controls 84
 cleaning up on exit 54
 ComposeLED 83
 ConsumeKeysOnComposeFail 82
 ConsumeLookupMods 81
 controls action types, table 153
 data structure 71
 DetectableAutorepeat 57
 determining the state of library controls 84
 determining which library controls are implemented 84
 effecting event delivery 83
EnabledControls 54
 enabling and disabling other controls 54
 for general keyboard mapping 68
ForceLatin1Lookup 81
GroupsWrap 69
IgnoreGroupLock 70
IgnoreLockMods 69
IgnoreNewKeyboards 83
InternalMods 70
keyboard 53
 keyboard use for physically-impaired persons 61
library controls masks 84
MouseKeys 59
MouseKeysAccel 59
overlays 58
PerKeyRepeat 56
querying 76
 repeat key behavior 56
RepeatKeys 56
SlowKeys 65
StickyKeys 67
 table listing all 72
 tracking changes to keyboard controls 78
 using the mouse from the keyboard 59
X library 81
Core Event, glossary entry 212

D

- Data structures** 11
 editing 11
 enlarging 11
 freeing 13
debounce_delay 75
Debugging 209
Detectable Auto-repeat, glossary entry 212
DetectableAutorepeat 53, 57, 72
Device feedback, types 197
Device identifier 10
Device specifications, matching with display specifications 9
Devices
 actions for generating DeviceButtonPress and DeviceButtonRelease 157
 actions simulating events from device valiators 158
 allocating, initializing and freeing data structures 202
 attaching Xkb actions to 197
 querying features for non-KeyClass devices 199
 querying for button actions 200
 querying indicator information 201
 setting features for non-KeyClass devices 203
 tracking changes to 206
Display, actions for changing active screen 152
Doodads 92, 95
 in sections 94
 indicator 95
 logo 95
 outline 95
 priority 95
 solid 95
 text 95
 types 95
Drawing a keyboard representation 96
DumbBells 64

E

Effective group 20
 Effective Group, glossary entry 212
 Effective mask 31
 Effective Mask, glossary entry 212
 Effective modifier mask 31
 Effective Modifier, glossary entry 213
 Effective modifiers 20
 enabled_ctrls 74
 EnabledControls 53, 54, 72
 Errors, protocol 9
 Events
 AccessXNotify 64
 base event code 14
 data structures 15
 interpreting key events 86
 MouseKeys 65
 overview 14
 RepeatKeys 65
 selecting for 15
 StickyKeys 65
 types 14
 types, table 14
 Explicit component masks, table 162
 Explicit components 162
 ExplicitAutoRepeat 162
 ExplicitBehavior 162
 ExplicitInterpret 162
 ExplicitKeyType1 162
 ExplicitKeyType2 162
 ExplicitKeyType3 162
 ExplicitKeyType4 162
 ExplicitVModMap 162
 Extension Device, glossary entry 213

F

Feedback, types 197
 Fonts, key label 92
 ForceLatin1Lookup 81

G

Geometry 3, 91
 adding elements to 105
 allocating and freeing components 109
 bounds, keyboard 92
 bounds, sections 94
 bounds, shapes 93
 computing the bounding box of a row 105
 computing the bounding box of a section 105
 computing the bounding box of a shape 104
 data structures 97
 data structures, diagram 97
 doodad types 95
 doodads 92, 95
 doodads in sections 94
 drawing a keyboard representation 96
 finding the overlay for a key 105
 functions for using 104
 getting from server 103
 key aliases 92
 key drawing order 94
 key label color 92
 key label font 92
 keyboard color 92
 keyboard with four sections, diagram 93

keys 94
 list of colors 92
 outlines 93
 outlines, diagram 104
 overlay keys 95
 overlay rows 95
 overlays 94
 priority 91, 94
 priority, doodads 95
 properties 92, 105
 rotated keyboard sections 91
 rotated keyboard sections, diagram 91
 rows 94
 rows in a section, diagram 94
 sections 92, 94
 shapes 92, 93
 top-level geometry description 91
 Global Keyboard Controls, glossary entry 213
 Grab group 21
 Grab modifiers 21
 Grab state 21
 Grab State, glossary entry 213
 Grabs
 passive, ignoring group locks 70
 Group Index, glossary entry 213
 Group, glossary entry 213
 Groups 20, 115, 116
 bindings for alternate group hints 192
 changing 23
 changing current state via key actions 144
 compatibility maps 168
 group action flags, table 145
 group action types, table 145
 group index constants 137
 handling illegal groups 69
 locking via actions 149
 normalizing groups into range 20, 133
 per-key group information 133
 symbolic group names 23
 treatment of out-of-range groups 133
 Groups Wrap Control, glossary entry 213
 GroupsWrap 53, 68, 69, 72, 74

H

Header files 6

I

IgnoreGroupLock 54, 68, 70, 72
 IgnoreLockMods 54, 68, 69, 72, 74
 IgnoreNewKeyboards 83
 Implicit support 86
 Indicator feedback 197
 Indicator Feedback, glossary entry 214
 Indicator map 35
 Indicator Map, glossary entry 214
 Indicator, glossary entry 214
 Indicators 3, 34
 allocating and freeing maps 45
 changing maps 42
 changing maps and state 41
 ComposeLED 83
 data structures 34
 effects of explicit changes on 41
 geometry, colors when lit and dark 95
 getting information about from server 39

- getting information by index 40
- getting information by name 40
- getting the state of 40
- how controls affect 39
- how groups affect 36
- how modifiers affect 37
- indicator drives keyboard 35
- keyboard drives indicator 35
- maps 35
- names 34
- querying names, maps and state 201
- tracking changes to state or map 44
- Initializing Xkb 7
- Input extension
 - attaching Xkb actions to devices 197
- Input Extension, glossary entry 214
- InternalMods 54, 68, 70, 72
- ISO9995 standard 115

- K**
- KbdFeedbackClass 48
- Key Action, glossary entry 214
- Key actions 140
 - independence of modifier state 116
- Key Alias, glossary entry 214
- Key aliases
 - geometry 92
 - names array 180
- Key Behavior, glossary entry 215
- Key events
 - interpreting 86
 - Xkb filtering out-of-range keycodes 187
- Key symbol map 132
- Key Symbol Map, glossary entry 215
- Key Type, glossary entry 213, 215
- Key types
 - ALPHABETIC 129
 - and shift levels 116
 - canonical 128
 - canonical key types, initializing 130
 - canonical, used in compatibility map 175
 - changing the number of levels in 131
 - copying key type data structures 131
 - getting from the server 130
 - KEYPAD 129
 - names 127
 - offset in symbol map 134
 - ONE_LEVEL 128
 - per-key key types indices 133
 - TWO_LEVEL 128
 - width (number of shift levels) 134
- Key types, example 127
- Key Width, glossary entry 213
- Key Width, Key Type Width, glossary entry 215
- Keyboard
 - components, server database 189
 - feedback 197
 - geometry 91
 - geometry sections 92
 - IgnoreNewKeyboards 83
 - names 179
 - replacing on the fly 186
 - symbolic name 92
 - unresponsiveness because of SlowKeys 62
- Keyboard Bells, glossary entry 215
- Keyboard Components, glossary entry 215
- Keyboard controls 53
 - for physically-impaired persons 61
 - tracking changes 78
- Keyboard description 2, 27
 - allocating and freeing 28
 - building from server database 192
 - changing 12
 - getting from server 28
 - updating library description 89
- Keyboard Feedback, glossary entry 215
- Keyboard Geometry Name, glossary entry 216
- Keyboard Geometry, glossary entry 216
- Keyboard mapping 115
 - client map 115
 - server map 115
 - shift levels and groups, diagram 116
- Keyboard state 19
 - base group 20, 211
 - base modifiers 20, 211
 - compatibility states 22
 - description 19
 - determining 23
 - effective group 20
 - effective modifiers 20
 - grab state 21
 - keysym groups 20
 - lookup state 21
 - modifiers 20
 - tracking 24
- Keyboard State, glossary entry 216
- KeyClass 10, 156, 197
- KeyCode Name, glossary entry 216
- KeyCode to string translation 81
- KeyCode, glossary entry 216
- Keycodes
 - actions for generating a different keycode for key 155
 - finding keysym bound to 88
 - keys which report more than one keycode 58
 - translating keycode to symbol and modifiers 90
 - Xkb filtering out-of-range key events 187
- Keymap
 - allocating and freeing 122, 123
 - changing map components 119
 - client map 125
 - functions 88
 - getting map components from the server 117
 - getting partial map components from the server, table 118
 - tracking changes to 121
- Keys
 - actions 140
 - aliases 92, 180
 - behavior 117, 160
 - behaviors, table 160
 - bindings hints 192
 - changing number of actions bound to key 159
 - changing number of groups and types for 136
 - changing the number of symbols bound to 137
 - finding keysym bound to 88
 - finding symbol for key with a particular state 88
 - for generating a different keycode for key 155
 - geometry 94
 - geometry, drawing order 94
 - getting per-key modifier map from server 138
 - getting the symbol map from the server 136

- label font and color 92
- obtaining key actions for keys from server 159
- obtaining key behaviors from the server 161
- offset in symbol map 134
- overlay geometry 95
- per-key group information 133
- per-key modifier map 138
- symbolic names 180
- types 126
- width (number of shift levels) 134
- Keysym** group 20, 115
- Keysym Group**, glossary entry 214
- Keysyms**
 - finding modifier set bound to keysym 88
 - finding symbol for key with a particular state 88
 - to string translation control 82
 - translating keycode to symbol and modifiers 90
- L**
- Latched Group**, glossary entry 216
- Latched Modifier**, glossary entry 216
- LatchToLock** 68
- Latin1 character set** lookup 81
- LED**, glossary entry 216
- Levels** 115, 116
 - and key types 116
 - changing the number in a key type 131
 - key types 126
 - names 127
- Linking with the Xkb extension** 6
- Locked Group**, glossary entry 216
- Locked Modifiers**, glossary entry 216
- Lookup group** 21
- Lookup modifiers** 21
- Lookup state** 21
- Lookup State**, glossary entry 217
- M**
- Major opcode** 7
- map** 3
- MappingNotify** 83, 86, 87, 174, 177, 187
- Messages**
 - actions for generating 154
 - detecting key action messages 154
- Modifier Definition**, glossary entry 217
- Modifier Key**, glossary entry 217
- Modifier**, glossary entry 217
- Modifiers** 20
 - action flags 144
 - action types, table 143
 - actions for changing the state of 142
 - bindings for modifier keys hints 192
 - changing the state via key actions 143
 - consume lookup modifiers control 81
 - effective mask 31
 - finding modifier set bound to keysym 88
 - forcing shift and lock to be consumed 82
 - getting per-key map from server 138
 - in actions to generate different keycode for key 156
 - inactive virtual modifiers 32
 - key action independent of 116
 - key types containing 126
 - locking via actions 149
 - masks 22
 - modifier definition 30
- names and masks** 30
- per-key modifier map** 138
- preserve field** 127
- preventing from being consumed** 127
- real** 30
- specifying which should be consumed by server** 70
- translating keycode to symbol and modifiers** 90
- virtual** 30
- virtual modifier server mapping** 163
- MotionNotify** 146
- Mouse**
 - using from the keyboard 59
- MouseKeys** 53, 59, 72
 - acceleration, diagram 61
 - changing button number simulated by mouse keys 148
- MouseKeysAccel** 53, 59, 72
 - absolute pointer motion 60
 - fields, table 59
 - relative pointer motion 60
- N**
- Names** 3
 - allocating and freeing symbolic names 185
 - changing symbolic names on server 182
 - getting keyboard description by component expression
 - names 193
 - getting symbolic names from server 182
 - shift level 127
 - symbolic 179
 - symbolic keyboard 92
 - symbolic names masks, table 181
 - tracking changes 184
 - types 127
- NewKeyboardNotify** 83
- Non-keyboard Extension Device**, glossary entry 217
- Normalizing groups** 20
- O**
- Outlines** 93
 - approximation 93
 - primary 93
- Outlines**, glossary entry 217
- Overlays**
 - controls 58
 - geometry keys 95
 - geometry rows 95
 - geometry, finding the overlay for a key 105
 - in geometry sections 94
 - Overlay1 and Overlay2 controls 53, 72
- P**
- PerKeyRepeat** 53, 56, 73, 76
- Physical Indicator Mask**, glossary entry 217
- Physical Symbol Keyboard Name**, glossary entry 217
- Pointer**
 - buttons 20
 - changing button number simulated by mouse keys 148
 - motion, absolute 60
 - motion, relative 60
 - moving via actions 146
 - pointer action types, table 146
 - pointer button action flags, table 148
 - pointer button action types, table 148
 - simulating pointer buttons via key actions 147
- Preserved Modifier**, glossary entry 217

Preserving modifiers from being consumed 127
P
 Priority
 doodads 95
 geometry 91
 sections 94
 Properties
 geometry 105
 Protocol errors 9
 added by Xkb 4

R
 Radio Group, glossary entry 218
 Radio groups 3, 160
 names 181
 Real Modifier, glossary entry 218
 Real modifiers 30
 RedirectInRange 69, 74, 134
 Remapping
 avoiding automatic by server 162
 repeat_delay 74
 repeat_interval 74
 Repeating keys
 controls 56
 detecting 57
 RepeatKeys 53, 56, 73
 Rows 94
 geometry 94
 overlay 95

S
 Sections 92, 94
 doodads in 94
 overlays 94
 priority 94
 Server
 avoiding automatic remapping by 162
 Server database 189
 changing map components 119
 class(member) form 190
 complete and partial entries 189
 component hints 191
 component names 190
 getting key types 130
 getting map components from 117
 getting partial map components from, table 118
 listing keyboard components 190
 obtaining virtual modifier bindings from 164
 virtual modifier definitions 163
 Server interaction with clients, diagram 166
 Server Internal Modifiers, glossary entry 218
 Server map 2
 allocating and freeing 123
 keyboard mapping 115, 139
 Shapes 92, 93
 Shift Level, glossary entry 218
 Shift levels 115, 116
 and key types 116
 changing the number of in a key type 131
 key types 126
 names 127
 SlowKeys 53, 65, 73
 acceptance delay 65, 75
 Standard, ISO9995 115
 State Field, glossary entry 218
 StickyKeys 53, 67, 73

automatically turning off 67
 locking a modifier 68
 Symbol Keyboard Name, glossary entry 218
 Symbolic Name, glossary entry 218
 Symbolic names 179

T
 Translating
 series of keysyms to string 81
 single keycode to string 81
 TwoKeys 67

V
 Valuator 158
 Valuator action 158
 Valuator, glossary entry 218
 Version, determining 6
 Virtual Modifier Mapping, glossary entry 218
 Virtual Modifier, glossary entry 218
 Virtual modifiers 30
 conventions for names 32
 data structure relationships, diagram 164
 effective mask 31
 example 32
 inactive 32
 key mapping 31
 master modifier definitions 31
 modifier definition 30
 names and masks 30
 obtaining bindings from server 164
 server mapping 163
 Visual bells, generating 47, 52

W
 Want and need components, table 194, 196
 WrapInRange 69, 74, 134

X
 X library controls 81
 X library functions affected by Xkb 87
 X server version required 1
 XChangeDeviceNotify 186
 XEvent 18
 Xkb
 attaching actions to input extension devices 197
 changes data structures 12
 compatibility map 166
 extension components 1
 extension library functions 4
 groups and shift levels 116
 implicit support 86
 keyboard extension support for keyboards 1
 keyboard mapping 115
 overall structure, diagram 2
 overview 1
 state, diagram 19
 X library functions affected 87
 Xkb client map, diagram 125
 Xkb events
 base event code 14
 data structures 15
 overview 14
 selecting for 15
 types 14
 types, table 14

Xkb extension
 disabling 8
 name 6
 Xkb server map, diagram 139
XKB.h 6
Xkb_RGAllowNone 161
XkbAccessXNotify 15, 64
XkbAccessXNotifyEvent 18, 64
XkbAction data structure 142
XkbActionCtrls macro 154
XkbActionMessage 15, 154
XkbActionMessageEvent 18, 155
XkbAddDeviceLedInfo() 202
XkbAddGeomColor() 106
XkbAddGeomDoodad() 108
XkbAddGeomKey() 107
XkbAddGeomKeyAlias() 106
XkbAddGeomOutline() 106
XkbAddGeomOverlay() 108
XkbAddGeomOverlayKey() 108
XkbAddGeomOverlayRow() 108
XkbAddGeomProperty() 106
XkbAddGeomRow() 107
XkbAddGeomSection() 107
XkbAddGeomShape() 107
XkbAddSymInterpret() 176
XkbAllocClientMap() 122
XkbAllocCompatMap() 178
XkbAllocControls() 80
XkbAllocDeviceInfo() 202
XkbAllocDeviceLedInfo() 202
XkbAllocGeomColors() 110
XkbAllocGeomDoodads() 113
XkbAllocGeometry() 114
XkbAllocGeomKeyAliases() 110
XkbAllocGeomKeys() 109
XkbAllocGeomOutlines() 109
XkbAllocGeomOverlayKeys() 113
XkbAllocGeomOverlayRows() 113
XkbAllocGeomOverlays() 112
XkbAllocGeomPoints() 111
XkbAllocGeomProps() 109
XkbAllocGeomRows() 112
XkbAllocGeomSectionDoodads() 113
XkbAllocGeomSections() 111
XkbAllocGeomShapes() 111
XkbAllocIndicatorMaps() 45
XkbAllocKeyboard() 28
XkbAllocNames() 185
XkbAllocServerMap() 123
XkbAlphabeticIndex canonical key type 128
XkbAnyAction data structure 142
XkbAnyEvent 15, 18
XkbApplyCompatMapToKey() 175
Xkb-aware client 3, 21
Xkb-aware Client, glossary entry 219
XkbAX_AnyFeedback macro 76
XkbAX_DumbBellFBMask 64
XkbAX_NeedFeedback macro 76
XkbAX_NeedOption macro 75
XkbAXN_AXKWarning 64
XkbAXN_BKAccept 64
XkbAXN_BKReject 64
XkbAXN_SKAccept 64
XkbAXN_SKPress 64
XkbAXN_SKReject 64
XkbAXN_SKRelease 64
XkbBehavior data structure 160
XkbBell() 49
XkbBellEvent() 50
XkbBellNotify 14, 47, 64
XkbBellNotifyEvent 18, 52
XkbBoundsRec 100
Xkb-capable client 3, 21
Xkb-capable Client, glossary entry 219
XkbChangeControls() 78
XkbChangeDeviceInfo() 207
XkbChangeEnabledControls() 54
XkbChangeIndicators() 44
XkbChangeMap() 121
XkbChangeNames() 183
XkbChangeTypesOfKey() 136
XkbClampIntoRange 69, 74, 134
XkbClientMapRec 126
XkbColorRec 100
XkbCompatMapNotify 14, 173, 177
XkbCompatMapNotifyEvent 18, 177
XkbCompatMapRec 168
XkbComponentListRec 191
XkbComponentNameRec 191
XkbComponentNamesRec 191
XkbComputeRowBounds() 105
XkbComputeSectionBounds() 105
XkbComputeShapeBounds() 104
XkbComputeShapeTop() 104
XkbControlsChangesRec 78
XkbControlsNotify 14, 62
XkbControlsNotifyEvent 18, 79
XkbControlsRec 72
 allocating and freeing 80
XkbCopyKeyType() 131
XkbCopyKeyTypes() 132
XkbCtrlsAction data structure 153
XkbDescRec 27
 component references 27
XkbDeviceBell() 49
XkbDeviceBellEvent() 50
XkbDeviceBtnAction data structure 157
XkbDeviceChangesRec 206
XkbDeviceInfoRec 198
XkbDeviceLedChangesRec 206
XkbDeviceLedInfoRec 198
XkbDeviceValuatorAction data structure 158
XkbDoodadRec 102
XkbEvent unified event type 18
XkbExtensionDeviceNotify 15, 204, 205, 207
XkbExtensionDeviceNotifyEvent 18, 206
XkbFindOverlayForKey() 105
XkbForceBell() 51
XkbForceDeviceBell() 51
XkbFreeClientMap() 123
XkbFreeCompatMap() 178
XkbFreeComponentList() 191
XkbFreeControls() 80
XkbFreeDeviceInfo() 203
XkbFreeGeomColors() 111
XkbFreeGeomDoodads() 114
XkbFreeGeometry() 114
XkbFreeGeomKeyAliases() 110
XkbFreeGeomKeys() 109

XkbFreeGeomOutlines() 109
 XkbFreeGeomOverlayKeys() 113
 XkbFreeGeomOverlayRows() 113
 XkbFreeGeomOverlays() 112
 XkbFreeGeomPoints() 111
 XkbFreeGeomProperties() 110
 XkbFreeGeomRows() 112
 XkbFreeGeomSections() 112
 XkbFreeGeomShapes() 111
 XkbFreeIndicatorMaps() 46
 XkbFreeKeyboard() 29
 XkbFreeNames() 185
 XkbFreeServerMap() 124
 XKgeom.h 6
 XkbGeometryRec 100
 XkbGetAccessXTimeout() 62
 XkbGetAutoRepeatRate() 57
 XkbGetAutoResetControls() 55
 XkbGetBounceKeysDelay() 66
 XkbGetCompatMap() 173
 XkbGetControls() 76
 XkbGetControlsChanges() 80
 XkbGetDetectableAutorepeat() 58
 XkbGetDeviceButtonActions() 200
 XkbGetDeviceInfo() 199
 XkbGetDeviceInfoChanges() 207
 XkbGetDeviceLedInfo() 201
 XkbGetGeometry() 103
 XkbGetIndicatorChanges() 45
 XkbGetIndicatorMap() 40
 XkbGetIndicatorState() 40
 XkbGetKeyActions() 159
 XkbGetKeyBehaviors() 161
 XkbGetKeyboard() 28, 196
 XkbGetKeyboardByName() 193
 XkbGetKeyExplicitComponents() 162
 XkbGetKeyModifierMap() 138
 XkbGetKeySyms() 136
 XkbGetKeyTypes() 130
 XkbGetKeyVirtualModMap() 165
 XkbGetMap() 117
 XkbGetNameChanges() 185
 XkbGetNamedGeometry() 104
 XkbGetNamedIndicator() 41
 XkbGetNames() 182
 XkbGetSlowKeysDelay() 66
 XkbGetState() 24
 XkbGetStickyKeysOptions() 68
 XkbGetUpdatedMap() 118
 XkbGetVirtualMods() 164
 XkbGetXlibControls() 84
 XkbGroupAction data structure 145
 XkbIgnoreExtension() 8
 XkbIM_LEDDrivesKB 35, 41
 XkbIM_NoAutomatic 35, 42
 XkbIM_NoExplicit 35, 41
 XkbIM_UseBase 37, 38
 XkbIM_UseCompat 38
 XkbIM_UseEffective 37, 38
 XkbIM_UseLatched 37, 38
 XkbIM_UseLocked 37, 38
 XkbIM_UseNone 37, 38
 XkbIndicatorChangesRec 43
 XkbIndicatorDoodadRec 103
 XkbIndicatorMapNotify 14, 44
 XkbIndicatorMapRec 35
 XkbIndicatorNotifyEvent 18, 44
 XkbIndicatorRec 34
 XkbIndicatorStateNotify 14, 44
 XkbInitCanonicalKeyTypes() 130
 XkbISOAction data structure 150
 XkbKB_Default 160
 XkbKB_Lock 161
 XkbKB_Overlay1 161
 XkbKB_Overlay2 161
 XkbKB_Permanent 161
 XkbKB_RadioGroup 161
 XkbKeyAction macro 141
 XkbKeyActionEntry macro 141
 XkbKeyActionsPtr macro 141
 XkbKeyAliasRec 100, 179
 XkbKeyCodeToKeysym() 88
 XkbKeyGroupInfo macro 134
 XkbKeyGroupsWidth macro 135
 XkbKeyGroupWidth macro 135
 XkbKeyHasActions macro 140
 XkbKeyNameRec 179
 XkbKeyNumActions macro 140
 XkbKeyNumGroups macro 134
 XkbKeyNumSyms macro 135
 XkbKeypadIndex canonical key type 128
 XkbKeyRec 101
 XkbKeySymEntry macro 135
 XkbKeySymsOffset macro 135
 XkbKeySymsPtr macro 135
 XkbKeysymToModifiers() 88
 XkbKeyType macro 133
 XkbKeyTypeIndex macro 133
 XkbKeyTypeRec 126
 XkbKeyTypesForCoreSymbols() 175
 XkbKTMMapEntryRec 126
 XkbLatchGroup() 23
 XkbLatchModifiers() 22
 XkbLC_AllControls 84
 XkbLC_AlphanumericKeys 192
 XkbLC_AlternateGroup 192
 XkbLC_AlwaysConsumeShiftAndLock 84
 XkbLC_BeepOnComposeFail 84
 XkbLC_ComposeLED 84
 XkbLC_ConsumeKeysOnComposeFail 84
 XkbLC_ConsumeLookupMods 84
 XkbLC_Default 192
 XkbLC_ForceLatin1Lookup 84
 XkbLC_FunctionKeys 192
 XkbLC_Hidden 192
 XkbLC_IgnoreNewKeyboards 84
 XkbLC_KeypadKeys 192
 XkbLC_ModifierKeys 192
 XkbLC_Partial 192
 XKBlib.h 6
 XkbLibraryVersion() 6
 XkbListComponents() 190
 XkbLockGroup() 23
 XkbLockModifiers() 22
 XkbLogoDoodadRec 103
 XkbLookupKeyBinding() 89
 XkbLookupKeySym() 88
 XkbMapChangesRec 120
 XkbMapNotify 14, 86, 87, 121, 177
 XkbMapNotifyEvent 18, 122

XkbMessageAction data structure 154
 XkbModAction data structure 143
 XkbModActionVMods macro 144
 XkbNameChangesRec 183
 XkbNamesNotify 14, 184
 XkbNamesNotifyEvent 18, 184
 XkbNamesRec 179
 XkbNewKeyboardNotify 14, 86, 186
 XkbNewKeyboardNotifyEvent 18, 187
 XkbNoteControlsChanges() 79
 XkbNoteDeviceChanges() 207
 XkbNoteIndicatorChanges() 45
 XkbNoteNameChanges() 184
 XkbOneLevelIndex canonical key type 128
 XkbOpenDisplay() 8
 XkbOutlineRec 100
 XkbOutOfRangeGroupInfo macro 134
 XkbOutOfRangeGroupNumber macro 134
 XkbOverlayKeyRec 101
 XkbOverlayRec 101
 XkbOverlayRowRec 101
 XkbPointRec 100
 XkbPropertyRec 100
 XkbPtrAction data structure 146
 XkbPtrActionX macro 147
 XkbPtrActionY macro 147
 XkbPtrBtnAction data structure 147
 XkbPtrDfltAction data structure 149
 XkbQueryExtension() 7
 XkbRedirectInRange 69, 74, 134
 XkbRedirectKeyAction data structure 155
 XkbRefreshKeyboardMapping() 89
 XkbResizeDeviceButtonActions() 203
 XkbResizeKeyActions() 159
 XkbResizeKeySyms() 137
 XkbResizeKeyType() 131
 XkbRowRec 101
 XkbSA_ActionMessage 142, 154
 XkbSA_AffectDfltBtn 148
 XkbSA_ClearLocks 144, 145
 XkbSA_DeviceBtn 142, 157
 XkbSA_DeviceValuator 142, 158
 XkbSA_DfltBtnAbsolute 149
 XkbSA_GroupAbsolute 145, 150
 XkbSA_IgnoreVal 158
 XkbSA_ISODfltlIsGroup 150, 151
 XkbSA_ISODNoAffectMods 151
 XkbSA_ISOLock 142, 150
 XkbSA_ISONoAffectCtrls 150, 151, 152
 XkbSA_ISONoAffectGroup 150, 151, 152
 XkbSA_ISONoAffectMods 150, 151
 XkbSA_ISONoAffectPtr 150, 151, 152
 XkbSA_LatchGroup 142, 145
 XkbSA_LatchMods 142, 143
 XkbSA_LatchToLock 144, 145
 XkbSA_LockControls 142, 153
 XkbSA_LockDeviceBtn 142
 XkbSA_LockDeviceBtn 157
 XkbSA_LockGroup 142, 145
 XkbSA_LockMods 142, 143
 XkbSA_LockNoLock 144, 148, 151, 153, 157
 XkbSA_LockNoUnlock 144, 148, 151, 153, 158
 XkbSA_LockPtrBtn 142, 148
 XkbSA_MessageGenKeyEvent 154
 XkbSA_MessageOnPress 154
 XkbSA_MessageOnRelease 154
 XkbSA_MoveAbsoluteX 146
 XkbSA_MoveAbsoluteY 147
 XkbSA_MovePtr 142, 146
 XkbSA_NoAcceleration 146
 XkbSA_NoAction 142
 XkbSA_PtrBtn 142
 XkbSA_PtrBtn 148
 XkbSA_RedirectKey 142, 155
 XkbSA_SetControls 142, 153
 XkbSA_SetGroup 142, 145
 XkbSA_SetMods 142, 143
 XkbSA_SetPtrDflt 142, 148
 XkbSA_SetValAbsolute 158
 XkbSA_SetValCenter 158
 XkbSA_SetValMax 158
 XkbSA_SetValMin 158
 XkbSA_SetValRelative 158
 XkbSA_SwitchAbsolute 152
 XkbSA_SwitchApplication 152
 XkbSA_SwitchScreen 142, 152
 XkbSA_UseDfltButton 148
 XkbSA_UseModMapMods 144, 151
 XkbSAActionSetCtrls macro 154
 XkbSAGroup macro 146
 XkbSAPtrDfltValue macro 149
 XkbSARedirectSetVMods macro 156
 XkbSARedirectSetVModsMask macro 156
 XkbSARedirectVMods macro 156
 XkbSARedirectVModsMask macro 156
 XkbSAScreen macro 152
 XkbSASetGroup macro 146
 XkbSASetPtrDfltValue macro 149
 XkbSASetScreen macro 153
 XkbSectionRec 102
 XkbSelectEventDetails() 17
 XkbSelectEvents mask constants 17
 XkbSelectEvents() 16
 XkbServerMapRec 140
 XkbSetAccessXTimeout() 63
 XkbSetAutoRepeatRate() 57
 XkbSetAutoResetControls() 55
 XkbSetBounceKeysDelay() 67
 XkbSetCompatMap() 176
 XkbSetControls() 77
 XkbSetDebuggingFlags() 209
 XkbSetDetectableAutorepeat() 58
 XkbSetDeviceButtonActions() 205
 XkbSetDeviceInfo() 204
 XkbSetIgnoreLockMods() 70
 XkbSetIndicatorMap() 42
 XkbSetMap() 119
 XkbSetModActionVMods macro 144
 XkbSetNamedIndicator() 43
 XkbSetName() 182
 XkbSetPtrActionX macro 147
 XkbSetPtrActionY macro 147
 XkbSetServerInternalMods() 71
 XkbSetSlowKeysDelay() 66
 XkbSetStickyKeysOptions() 68
 XkbSetXlibControls() 84
 XkbShapeDoodadRec 102
 XkbShapeRec 100
 XkbSI_AllOf 171
 XkbSI_AnyOf 171

XkbSI_AnyOfOrNone 171
XkbSI_Exactly 171
XkbSI_NoneOf 171
XkbStateNotify 14, 24, 65
XkbStateNotify event detail masks 24
XkbStateNotifyEvent 18, 25
XkbStateRec 24
Xkbstr.h 6
XkbSwitchScreenAction data structure 152
XkbSymInterpretRec 171
XkbSymMapRec 132
XkbTextDoodadRec 103
XkbTranslateKeyCode() 90
XkbTranslateKeySym() 89
XkbTwoLevelIndex canonical key type 128
Xkb-unaware client 4, 21
Xkb-unaware Client, glossary entry 219
XkbUpdateMapFromCore() 174
XkbUseCoreKbd 10, 15
XkbVirtualModsToReal() 32
XkbWrapIntoRange 69, 74, 134
XkbXIDfltID 197
XkbXlibControlsImplemented() 84
XKeycodeToKeysym(), Xkb modifications 87
XKeysymToKeycode(), Xkb modifications 87
Xlib version required 1
XLookupKeysym(), Xkb modifications 87
XLookupString() 81
 function which is equivalent, XkbLookupKeyBinding() 89
 Xkb modifications 87
XMappingNotify 186
XRebindKeysym(), Xkb modifications 88
XRefreshKeyboardMapping()
 function which is equivalent, XkbRefreshKeyboardMap-
 ping() 89
 Xkb modifications 87