How to further enhance XKB configuration

Kamil Toman
Ivan U. Pascal

25 November 2002

Abstract

This guide is aimed to relieve one's labour to create a new (internationalized) keyboard layout. Unlike other documents this guide accents the keymap developer's point of view.

Table of Contents

Overview

The developer of a new layout should read the xkb protocol specification (The X Keyboard Extension: Protocol Specification [http://www.x.org/docs/XKB/XKBproto.pdf]) at least to clarify for himself some xkb-specific terms used in this document and elsewhere in xkb configuration. Also it shows wise to understand how the X server and a client digest their keyboard inputs (with and without xkb).

A useful source is also Ivan Pascal's text about xkb configuration [http://www.tsu.ru/~pascal/en/xkb] often referenced throughout this document.

Note that this document covers only enhancements which are to be made to XFree86 version 4.3 and X11R6.7.0 and above.

The Basics

At the startup (or at later at user's command) X server starts its xkb keyboard module extension and reads data from a compiled configuration file.

This compiled configuration file is prepared by the program **xkbcomp** which behaves altogether as an ordinary compiler (see `man xkbcomp`). Its input are human readable xkb configuration files which are verified and then composed into a useful xkb configuration. Users don't need to mess with **xkbcomp** themselves, for them it is invisible. Usually, it is started upon X server startup.

As you probably already know, the xkb configuration consists of five main modules:

Keycodes	Tables that defines translation from keyboard scan codes into reasonable symbolic names, maximum, minimum legal keycodes, symbolic aliases and description of physically present LED-indicators. The primary sence of this component is to allow definitions of maps of symbols (see below) to be independent of physical keyboard scancodes. There are two main naming conventions for symbolic names (always four bytes long): • names which express some traditional meaning like <SPCE> (stands for space bar) or • names which express some relative positioning on a keyboard, for example <AE01> (an exclamation mark on US keyboards), on the right there are keys <AE02>, <AE03> etc.
Types	Types describe how the produced key is changed by active modifiers (like Shift, Control, Alt, ...). There are several predefined types which cover most of used combinations.
Compat	Compatibility component defines internal behaviour of modifiers. Using compat component you can assign various actions (elaborately described in xkb specification) to key events. This is also the place where LED-indicators behaviour is defined.
Symbols	For i18n purposes, this is the most important table. It defines what values (=symbols) are assigned to what keycodes (represented by their symbolic name, see above). There may be defined more than one value for each key and then it depends on a key type and on modifiers state (respective compat component) which value will be the resulting one.
Geometry	Geometry files aren't used by xkb itself but they may be used by some external programs to depict a keyboard image.

All these components have the files located in xkb configuration tree in subdirectories with the same names (usually in `/usr/lib/X11/xkb`).

Enhancing XKB Configuration

Most of xkb enhancements concerns a need to define new output symbols for the some input key events. In other words, a need to define a new symbol map (for a new language, standard or just to feel more comfortable when typing text).

What do you need to do? Generally, you have to define following things:

- the map of symbols itself
- the rules to allow users to select the new mapping
- the description of the new layout

First of all, it is good to go through existing layouts and to examine them if there is something you could easily adjust to fit your needs. Even if there is nothing similar you may get some ideas about basic concepts and used tricks.

Levels And Groups

Since XFree86 4.3.0 and X11R6.7.0 you can use *multi-layout* concept of xkb configuration. Though it is still in boundaries of xkb protocol and general ideas, the keymap designer must obey new rules when creating new maps. In exchange we get a more powerful and cleaner configuration system.

Remember that it is the application which must decide which symbol matches which keycode according to effective modifier state. The X server itself sends only an input event message to. Of course, usually the general interpretation is processed by Xlib, Xaw, Motif, Qt, Gtk and similar libraries. The X server only supplies its mapping table (usually upon an application startup).

You can think of the X server's symbol table as of a irregular table where each keycode has its row and where each combination of modifiers determines exactly one column. The resulting cell then gives the proper symbolic value. Not all keycodes need to bind different values for different combination of modifiers. <ENTER> key, for instance, usually doesn't depend on any modifiers so it its row has only one column defined.

Note that in XKB there is no prior assumption that certain modifiers are bound to certain columns. By editing proper files (see keytypes) this mapping can be changed as well.

Unlike the original X protocol the XKB approach is far more flexible. It is comfortable to add one additional XKB term - group. You can think of a group as of a vector of columns per each keycode (naturally the dimension of this vector may differ for different keycodes). What is it good for? The group is not very useful unless you intend to use more than one logically different set of symbols (like more than one alphabet) defined in a single mapping table. But then, the group has a natural meaning - each symbol set has its own group and changing it means selecting a different one. XKB approach allows up to four different groups. The columns inside each group are called (shift) levels. The X server knows the current group and reports it together with modifier set and with a keycode in key events.

To sum it up:

- for each keycode XKB keyboard map contains up to four one-dimensional tables - groups (logically different symbol sets)
- for each group of a keycode XKB keyboard map contains some columns - shift levels (values reached by combinations of Shift, Ctrl, Alt, ... modifiers)
- different keycodes can have different number of groups
- different groups of one keycode can have different number of shift levels
- the current group number is tracked by X server

It is clear that if you sanely define levels, groups and sanely bind modifiers and associated actions you can have simultaneously loaded up to four different symbol sets where each of them would reside in its own group.

The multi-layout concept provides a facility to manipulate xkb groups and symbol definitions in a way that allows almost arbitrary composition of predefined symbol tables. To keep it fully functional you have to:

- define all symbols only in the first group
- (re)define any modifiers with extra care to avoid strange (anisometric) behaviour

Defining New Layouts

See Some Words About XKB internals [http://www.tsu.ru/~pascal/en/xkb/internals.html] for explanation of used xkb terms and problems addressed by XKB extension.

See Common notes about XKB configuration files language [http://www.tsu.ru/~pascal/en/xkb/gram-common.html] for more precise explanation of syntax of xkb configuration files.

Predefined XKB Symbol Sets

If you are about to define some European symbol map extension, you might want to use on of four predefined latin alphabet layouts.

Okay, let's assume you want extend an existing keymap and you want to override a few keys. Let's take a simple U.K. keyboard as an example (defined in `pc/gb`):

```
partial default alphanumeric_keys
xkb_symbols "basic" {
  include "pc/latin"

  name[Group1]="Great Britain";

  key <AE02>  { [         2,   quotedbl,  twosuperior,    oneeighth ] };
  key <AE03>  { [         3,   sterling, threesuperior,    sterling ] };
  key <AC11>  { [apostrophe,         at, dead_circumflex, dead_caron] };
  key <TLDE>  { [     grave,    notsign,          bar,          bar ] };
  key <BKSL>  { [numbersign, asciitilde,   dead_grave,   dead_breve ] };
  key <RALT>  { type[Group1]="TWO_LEVEL",
                [ ISO_Level3_Shift, Multi_key ]   };

  modifier_map Mod5   { <RALT> };
};
```

It defines a new layout in `basic` variant as an extension of common latin alphabet layout. The layout (symbol set) name is set to "Great Britain". Then there are redefinitions of a few keycodes and a modifiers binding. As you can see the number of shift levels is the same for <AE02>, <AE03>, <AC11>, <TLDE> and <BKSL> keys but it differs from number of shift levels of <RALT>.

Note that the <RALT> key itself is a binding key for Mod5 and that it serves like a shift modifier for LevelThree, together with Shift as a multi-key. It is a good habit to respect this rule in a new similar layout.

Okay, you could now define more variants of your new layout besides `basic` simply by including (augmenting/overriding/...) the basic definition and altering what may be needed.

Key Types

The differences in the number of columns (shift levels) are caused by a different types of keys (see the types definition in section basics). Most keycodes have implicitly set the keytype in the included “`pc/latin`” file to “`FOUR_LEVEL_ALPHABETIC`”. The only exception is <RALT> keycode which is explicitly set “`TWO_LEVEL`” keytype.

All those names refer to pre-defined shift level schemes. Usually you can choose a suitable shift level scheme from `default` types scheme list in proper xkb component's subdirectory.

The most used schemes are:

ONE_LEVEL	The key does not depend on any modifiers. The symbol from first level is always chosen.
TWO_LEVEL	The key uses a modifier Shift and may have two possible values. The second level may be chosen by Shift modifier. If Lock modifier (usually Caps-lock) applies the symbol is further processed using system-specific capitalization rules. If both Shift+Lock modifier apply the symbol from the second level is taken and capitalization rules are applied (and usually have no effect).
ALPHABETIC	The key uses modifiers Shift and Lock. It may have two possible values. The second level may be chosen by Shift modifier. When Lock modifier applies, the symbol from the first level is taken and further processed using system-specific capitalization rules. If both Shift+Lock modifier apply the symbol from the first level is taken and no capitalization rules applied. This is often called shift-cancels-caps behaviour.
THREE_LEVEL	Is the same as TWO_LEVEL but it considers an extra modifier - LevelThree which can be used to gain the symbol value from the third level. If both Shift+LevelThree modifiers apply the value from the third level is also taken. As in TWO_LEVEL, the Lock modifier doesn't influence the resulting level. Only Shift and LevelThree are taken into that consideration. If the Lock modifier is active capitalization rules are applied on the resulting symbol.
FOUR_LEVEL	Is the same as THREE_LEVEL but unlike LEVEL_THREE if both Shift+LevelThree modifiers apply the symbol is taken from the fourth level.
FOUR_LEVEL_ALPHABETIC	Is similar to FOUR_LEVEL but also defines shift-cancels-caps behaviour as in ALPHABETIC. If Lock+LevelThree apply the symbol from the third level is taken and the capitalization rules are applied. If Lock+Shift

	+LevelThree apply the symbol from the third level is taken and no capitalization rules are applied.
KEYPAD	As the name suggest this scheme is primarily used for numeric keypads. The scheme considers two modifiers - Shift and NumLock. If none of modifiers applies the symbol from the first level is taken. If either Shift or NumLock modifiers apply the symbol from the second level is taken. If both Shift+NumLock modifiers apply the symbol from the first level is taken. Again, shift-cancels-caps variant.
FOUR_LEVEL_KEYPAD	Is similar to KEYPAD scheme but considers also LevelThree modifier. If LevelThree modifier applies the symbol from the third level is taken. If Shift+LevelThree or NumLock+LevelThree apply the symbol from the fourth level is taken. If all Shift+NumLock+LevelThree modifiers apply the symbol from the third level is taken. This also, shift-cancels-caps variant.

Besides that, there are several schemes for special purposes:

PC_BREAK	It is similar to TWO_LEVEL scheme but it considers the Control modifier rather than Shift. That means, the symbol from the second level is chosen by Control rather than by Shift.
PC_SYSRQ	It is similar to TWO_LEVEL scheme but it considers the Alt modifier rather than Shift. That means, the symbol from the second level is chosen by Alt rather than by Shift.
CTRL +ALT	The key uses modifiers Alt and Control. It may have two possible values. If only one modifier (Alt or Control) applies the symbol from the first level is chosen. Only if both Alt+Control modifiers apply the symbol from the second level is chosen.
SHIFT +ALT	The key uses modifiers Shift and Alt. It may have two possible values. If only one modifier (Alt or Shift) applies the symbol from the first level is chosen. Only if both Alt+Shift modifiers apply the symbol from the second level is chosen.

If needed, special `caps` schemes may be used. They redefine the standard behaviour of all `*ALPHABETIC` types. The layouts (maps of symbols) with keys defined in respective types then automatically change their behaviour accordingly. Possible redefinitions are:

- internal
- internal_nocancel
- shift
- shift_nocancel

None of these schemes should be used directly. They are defined merely for `'caps:'` xkb options (used to globally change the layouts behaviour).

Don't alter any of existing key types. If you need a different behaviour create a new one.

More On Definitions Of Types

When the XKB software deals with a separate type description it gets a complete list of modifiers that should be taken into account from the `'modifiers=<list of modifiers>'` list and expects that a set of `'map[<combination of modifiers>]=<list of modifiers>'` instructions that contain the mapping for each combination of modifiers mentioned in that list. Modifiers that are not explicitly listed are NOT taken into account when the resulting shift level is computed. If some combination is omitted the program (subroutine) should choose the first level for this combination (a quite reasonable behavior).

Lets consider an example with two modifiers ModOne and ModTwo:

```
type "..." {
    modifiers = ModOne+ModTwo;
    map[None] = Level1;
    map[ModOne] = Level2;
};
```

In this case the map statements for ModTwo only and ModOne+ModTwo are omitted. It means that if the ModTwo is active the subroutine can't found explicit mapping for such combination an will use the *default level* i.e. Level1.

But in the case the type described as:

```
type "..." {
    modifiers = ModOne;
    map[None] = Level1;
    map[ModOne] = Level2;
};
```

the ModTwo will not be taken into account and the resulting level depends on the ModOne state only. That means, ModTwo alone produces the Level1 but the combination ModOne+ModTwo produces the Level2 as well as ModOne alone.

What does it mean if the second modifier is the Lock? It means that in the first case (the Lock itself is included in the list of modifiers but combinations with this modifier aren't mentioned in the map statements) the internal capitalization rules will be applied to the symbol from the first level. But in the second case the capitalization will be applied to the symbol chosen accordingly to he first modifier - and this can be the symbol from the first as well as from the second level.

Usually, all modifiers introduced in `'modifiers=<list of modifiers>'` list are used for shift level calculation and then discarded. Sometimes this is not desirable. If you want to use a modifier for shift level calculation but you don't want to discard it, you may list in `'preserve[<combination of modifiers>]=<list of modifiers>'`. That means, for a given combination all listed modifiers will be preserved. If the Lock modifier is preserved then the resulting symbol is passed to internal capitalization routine regardless whether it has been used for a shift level calculation or not.

Any key type description can use both real and virtual modifiers. Since real modifiers always have standard names it is not necessary to explicitly declare them.

Virtual modifiers can have arbitrary names and can be declared (prior using them) directly in key type definition:

```
virtual_modifiers <comma-separated list of modifiers>  ;
```

as seen in for example `basic`, `pc` or `mousekeys` key type definitions.

Rules

Once you are finished with your symbol map you need to add it to rules file. The rules file describes how all the five basic keycodes, types, compat, symbols and geometry components should be composed to give a sensible resulting xkb configuration.

The main advantage of rules over formerly used keymaps is a possibility to simply parameterize (once) fixed patterns of configurations and thus to elegantly allow substitutions of various local configurations into predefined templates.

A pattern in a rules file (often located in `/usr/lib/X11/xkb/rules`) can be parameterized with four other arguments: `Model`, `Layout`, `Variant` and `Options`. For most cases parameters `model` and `layout` should be sufficient for choosing a functional keyboard mapping.

The rules file itself is composed of pattern lines and lines with rules. The pattern line starts with an exclamation mark ('`!`') and describes how will the xkb interpret the following lines (rules). A sample rules file looks like this:

```
! model                 = keycodes
  macintosh_old                = macintosh
  ...
  *                  = xorg

! model                 = symbols
  hp                  = +inet(%m)
  microsoftpro                = +inet(%m)
  geniuscomfy                = +inet(%m)

! model     layout[1]         = symbols
  macintosh us         = macintosh/us%(v[1])
  *            *               =       pc/pc(%m)+pc/%l[1]%(v[1])

! model     layout[2]         = symbols
  macintosh     us              = +macintosh/us[2]%(v[2]):2
  *  *               = +pc/%l[2]%(v[2]):2

! option   = types
  caps:internal   = +caps(internal)
  caps:internal_nocancel = +caps(internal_nocancel)
```

Each rule defines what certain combination of values on the left side of equal sign ('=') results in. For example a (keyboard) model `macintosh_old` instructs xkb to take definitions of keycodes from file `keycodes/macintosh` while the rest of models (represented by a wild card '*') instructs it to take them from file `keycodes/xorg`. The wild card represents all possible values on the left side which were not found in any of the previous rules. The more specialized (more complete) rules have higher precedence than general ones, i.e. the more general rules supply reasonable default values.

As you can see some lines contain substitution parameters - the parameters preceded by the percent sign ('%'). The first alphabetical character after the percent sign expands to the value which has been found on the left side. For example `+%l%(v)` expands into `+cz(bksl)` if the respective values on the left side were `cz` layout in its `bksl` variant. More, if the layout resp. variant parameter is followed by a pair of brackets ('`[`', '`]`') it means that xkb should *place the layout resp. variant into specified xkb group*. If the brackets are omitted the first group is the default value.

So the second block of rules enhances symbol definitions for some particular keyboard models with extra keys (for internet, multimedia, ...) . Other models are left intact. Similarly, the last block overrides some key type definitions, so the common global behaviour ''shift cancels caps'' or ''shift doesn't cancel caps'' can be selected. The rest of rules produces special symbols for each variant `us` layout of `macintosh` keyboard and standard pc symbols in appropriate variants as a default.

Descriptive Files of Rules

Now you just need to add a detailed description to `<rules>.xml` description file so the other users (and external programs which often parse this file) know what is your work about.

Old Descriptive Files

The formerly used descriptive files were named `<rules>.lst` Its structure is very simple and quite self descriptive but such simplicity had also some cavities, for example there was no way how to describe local variants of layouts and there were problems with the localization of descriptions. To preserve compatibility with some older programs, new XML descriptive files can be converted to old format '.lst'.

For each parameter of rules file should be described its meaning. For the rules file described above the `.lst` file could look like:

```
! model
  pc104  Generic 104-key PC
  microsoft Microsoft Natural
  pc98  PC-98xx Series
  macintosh     Original Macintosh
  ...

! layout
  us  U.S. English
  cz  Czech
  de  German
  ...

! option
  caps:internal   uses internal capitalization. Shift cancels Caps
```

```
caps:internal_nocancel uses internal capitalization. Shift doesn't cancel Caps
```

And that should be it. Enjoy creating your own xkb mapping.