An Unreliable Guide to XKB Configuration

Doug Palmer*

October 11, 2004

> Oh yes. I've been through there on my trip around the image. The giant vaulted Klein bottles covered with mosaics of other, different, Klein bottles … the rows of gargoyles on the roof, each holding a sign reading "See Previous Gargoyle" … the little food stands around the base, where they sell you food stand vouchers, redeemable for food stand vouchers at all food stands except this one … the hall of the penitents … the giant Romanesque stained glass windows, built out of thousands of tiny LooksLike blocks, lit from behind by the radiance of the great Aka …
>
> Truly one of the architectural wonders of our age. I've been there alright. And I bought postcards.

Steve Taylor, talking about software completely unrelated to XKB

> Before you read this, please understand that I never wanted to write this document, being grossly under-qualified, but I always wanted to read it, and this was the only way.

Paul "Rusty" Russel

Contents

*mailto:doug@charvolant.org http://www.charvolant.org/~doug

1 Introduction

This document is one of those annoying things that you write because, try as you might, you just can't find any documentation on something.[1] It all started when I wanted to use my beloved "Happy Hacking Keyboard"[4] with the A^+ language[1]. After a little futile hacking, I realised that, in order to use the funny A^+ symbols, I would need to properly map the HHK keyboard so that I could use the various meta-keys. Thus began my descent into hell.

The XKB protocol and library are quite well documented. However, there appears to be next to no information on the various files contained in `/usr/X11R6/lib/X11/xkb/`. These files are used by XFree86[8] (and others) to provide XKB information. They appear to follow, more or less, the conventions described in the XKB protocol document[3] but there are a number of bits and pieces that don't have any clear meaning.

This document is a result of hacking through the thickets of the XKB configuration files, the protocol documentation and the (undocumented) source of **xkbcomp** in a bout of reverse-engineering.[2] I can't pretend that it's complete or even correct. Any comments or corrections are most welcome.

2 The Basics

The XKB configuration has been decomposed into a number of components. The basic idea is that you can adopt a mix-and-match approach to building a keyboard configuration. Some pretty sophisticated inclusion and augmentation rules allow a component to be built from a basic set-up and a number of modifications for odd keyboard layouts and national peculiarities. The basic components are:

key codes A translation of the scan codes from the keyboard into a suitable symbolic form.

key symbols A translation of symbolic key codes into actual symbols, such as an **A** or an **ä**.

compatibility map A specification of what actions various special-purpose keys produce.

type A specification of what various shift combinations produce.

geometry A description of the physical layout of a keyboard.

[1] Which should teach me something. Halfway through writing this, I discovered Ivan Pascal's documentation.[6] Still, the more the merrier. In the meantime, I've stolen shamelessly from Ivan's documentation.

[2] Undocumented configuration files. Undocumented code. Open-source is all very nice, but this represents a pretty high barrier to entry. It's been pointed out before that a non-trivial open-source project without adequate documentation might as well be close-source.[2]

There are a number of other components: rules, semantics, keymaps that are essentially ways of packaging the main components into more usable collections.

2.1 Modifier Keys

The *modifier keys* are those keys, like Shift, Control or Alt that are used to change the meaning of other keys. Modifier keys can be combined, to give combinations like Control+Shift+Alt. Handling extended combinations of modifier keys is, to a certain extent, what makes XKB so complex.

At the base level, XKB recognises eight modifier keys: the explicitly named Control, Shift and Lock keys and the generic Mod1–Mod5 keys. These keys correspond to the keys in the core X protocol and need to be there so that older programs can understand what's going on. Keys like the ubiquitous Alt keys are mapped onto one of the Mod keys.

The basic modifier keys are all very well but it would be handy to be able to introduce a level of abstraction, so that you can talk about modifier keys by function, rather than by explicit key name. XKB allows the use of *virtual modifier* keys, where a basic modifier key (or combination) is mapped onto a named virtual modifier. Virtual modifiers can then be used to describe the behaviour of the keyboard, decoupling the exact physical capabilities of the keyboard you are using from the sort of characters that you want to type. The **types** and **compat** components are largely responsible for handling this side of things.

2.2 Levels and Groups

Once the modifier keys have been set up, it now becomes possible to have different combinations of keys produce different characters from the keyboard. In theory, you could do more or less anything that you wanted to here. However, a complete free for all approach would make combining different components very difficult. To provide some structure, XKB uses two organising principles: levels and groups.

A *level* represents the sort of thing that a shift key is expected to do. Normally, when you press the key marked A you would expect an **a** character to appear. If you hold the shift key down and press the same key, you would expect an **A** to appear instead.

The **a** and **A** characters are clearly related. And this is what you would expect a level shift to do, provide you with some sort of variation of the same thing, capitalisation in this case. Combinations such as **8** and ***** are less obvious, but have a sort of honorary status, since they appear together on both Qwerty and Dvorak layouts.

For most purposes, two levels are enough, shifted and non-shifted. There is a fairly standard way of invoking different levels, which is to press the Shift or Lock keys. This nice state of affairs is complicated by keys that don't have multiple levels — such as the Return key — and keys that often have more complex levels — such as the numeric keypad. But more on that later.

In contrast to levels, which are relatively straightforward, *groups* are a more slippery concept. The basic idea behind groups is that, sometimes, you want to shift the entire keyboard over to some other character set, so that you can access characters not usually considered part of a standard keyboard, such as **Æ**, **ß**, ¶ or **£**. The need for these characters isn't so obvious for an English speaker but people who write in languages with richer character sets than English, such as Czech, Arabic or Chinese, not to mention those writing mathematics or using languages such as A^{+}, will immediately see the need for multiple groups.

The keys needed to shift groups are less obvious than those needed to shift levels, since there isn't any immediate equivalent to the Shift key and the Alt key is often given other duties. The standard XKB configuration files supply a number of possible combinations that could be used and it's up to the user to choose what they want.

Within each group, there can be multiple levels, with each level reflecting the suitable shifts for the characters in the group. For example, **æ** would be shifted to **Æ**, as shown in figure 1

2.3 Key Codes

At the bottom of the XKB food chain are *key codes*. Raw key codes are the numeric codes generated by a particular keyboard to indicate that a key has been pressed or released. The X system generates two events

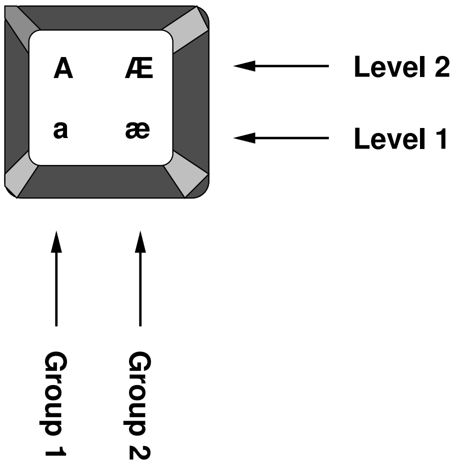

Figure 1: Levels and Groups on a Single Key

for each key press: one to indicate that the key has been pressed, one to indicate that it has been released.[3] In both cases, the key code indicates which key has been pressed or released.

Hardware designers are free to choose whatever numbers they like for keycodes. So, left to themselves, raw keycodes tend to obscure the fact that most keyboards, no matter who made them, have a similar layout. The **keycodes** component of XKB allows symbolic names to be assigned to the various key codes. These symbolic names can then be used to look up similar keyboard layouts in the **symbols** component (see below).

2.4 Key Symbols

Key symbols are the actual characters or glyphs that pressing a key produces. A symbols map maps symbolic key codes (see above) onto the appropriate symbol. Symbols map also handle specifications for which keys are modifier keys.

Each symbol has a name, defined as part of the X protocol, with a few extra added by XKB.[7, 3]. The actual symbol names are taken, as far as I can see from the `/usr/X11R6/include/X11/keysymdefh` file. Each name is the name in this list, with the initial `XK_` stripped off. The name of simple symbols, such as **a** is `a`. The name of more complex symbols, such as **ß**, is spelt out as `ssharp`. The XKB protocol specification also spells out a number of odd key symbols in appendix C.[3]

Groups and levels (see section 2.2) alter the meaning of the keys on the keyboard. The symbol map is, therefore, similar to a matrix. Symbol maps usually list multiple symbols for each key, with the group and level being used to look up the appropriate symbol.

3 Choosing an XKB Configuration

The most likely approach to configuring XKB is to use the already existing configuration files, and you want to stick components together so that you have your own private eccentricities.

Ivan Pascal has a web page that describes how to do this in various levels of detail.[6] Like Pascal, I'm going to assume that you are using XFree86 and that you are going to be using the `/etc/X11/XF86Config-4` configuration file to set things up. People using other X11 implementations will need to find the appropriate file for themselves.

The first thing to learn is how to access the definitions in the configuration files. To start with, the configuration files are in various subdirectories named after the type of components that they represent.

[3] The keyboard itself may generate up to six events for each key press. These events are pre-processed and gathered together before being passed onto XKB.

Since it will always be obvious when you are using, say, a **compat** component instead of a **keycodes** component, it is unnecessary to explicitly name the component type in the name.

In the subdirectories, there are configuration files and further subdirectories. To access a configuration file, you just use its name, eg. **xfree86**. If the file is in a subdirectory, then the subdirectory also has to be named, eg. **sgi/iris**.

Each file can contain multiple configuration variants. So, for example, in the **xfree86** keycodes file, there are several variants: **xfree86**, **basic**, **102**, **jp106**, **jp109usb** and **abnt2**. One of these variants will be marked as the default, which is what you get if you don't specify an explicit variant. Otherwise, you have to include the variant name in brackets, eg. **xfree86(pc102)**.

Once you have a basic component, you can extend that component by adding additional components to it. These additional components modify or augment the meaning of the basic component. The notation here is that you use `+` to *override* any existing definitions and `|` to *augment*. An overridden definition always replaces an existing one, an augmented definition is only used if a definition doesn't already exist. Hence, to specify **symbols** where you are using a US keyboard on a 101-key PC keyboard, but you want to swap the caps-lock and left-control keys, you could say **us(pc101)+ctrl(swapcaps)**.

3.1 Doing it the Easy Way

Rather than worry too much about the exact combinations of basic components and their extensions, XKB uses the **rules** component to allow suitable combinations to be picked. The rules allow a few key words to be specified in the `XF86Config-4` file and the results are translated into a suitable XKB configuration.

In the `XF86Config-4` file, there will be a section which looks something like this:

```
Section "InputDevice"
        Identifier  "Keyboard0"
        Driver      "keyboard"

        Option  "XkbRules"      "xfree86"
        Option  "XkbModel"      "pc104"
        Option  "XkbLayout"     "us"
        Option  "XkbVariant"    "basic"
        Option  "XkbOptions"    "grp:menu_toggle"
EndSection
```

The `Option` statements contain the interesting bits. For XFree86, the file `/usr/X11R6/lib/X11/xkb/rules/xfree86.lst` contains descriptions of the various elements. The various elements that you can have are:

XkbRules The set of rules that should be followed to get a configuration. For XFree86, this should — pretty obviously – be `xfree86`. Other options are `sgi` and `sun`.

XkbModel The type of keyboard that you are using. For example, `pc104` is a generic 104-key PC keyboard.

XkbLayout The keyboard layout that you want. Layouts usually express national variations (eg. `de` for a German keyboard) but they can also be used for alternate keyboard layouts (eg. `dvorak` for a Dvorak layout) or other oddities. Users of US keyboards have two options: `us` for plain vanilla US keys and `en_US` for a keyboard with a group of interesting additional characters.

XkbVariant Any minor variants on the general layout. This is usually set to `basic`.

XkbOptions Any specific options. Options usually relate to how to shift groups, the position of the control keys and how to indicate that the group has been changed. For example, the option `grp:toggle` means that the Right-Alt key toggles between groups.

Note that, if you want to use more than one group, you have to specify a group-change option.

3.2 Doing it the Slightly Cruder Way

The **keymap** component provides a way of setting up basic national variants, without the level of detail required by the **rules** component. An extended PC keyboard (ie. with the Windows keys, etc.) is generally assumed and the national symbol sets are mapped onto that keyboard.

To use key maps, use the `XkbKeymap` option in the `XF86Config-4` file to specify the national set. For example, the following setup configures the keyboard for Belgian use:

```
Section "InputDevice"
        Identifier  "Keyboard0"
        Driver      "keyboard"

        Option  "XkbKeymap"      "be"
EndSection
```

3.3 Doing it the Hard Way

Doing it the hard way involves specifying the five major XKB components explicitly. If you do things this way, then you need to explicitly include any special options or variants that you might want to use. The benefit to doing things this way, of course, is that you have a lot of control over the exact keyboard configuration.

To specify the major components, you will need to explicitly state the following options: `XkbKeycodes`, `XkbTypes`, `XkbCompat`, `XkbSymbols` and XkbGeometry. As an example, you might have:

```
Section "InputDevice"
        Identifier  "Keyboard0"
        Driver      "keyboard"

        Option  "XkbKeycodes"      "xfree86"
        Option  "XkbTypes"         "default"
        Option  "XkbCompat"        "basic+pc+iso9995+norepeat"
        Option  "XkbSymbols"       "en_US(pc104)+dk+ctrl(swapcaps)+group(switch)"
        Option  "XkbGeometry"      "pc(pc104)"
EndSection
```

What all this means is:

xfree86 Use the standard XFree86 interpretation of keyboard codes. This essentially corresponds to a standard PC keyboard.

default Use the default types (interpretation of level shifts). The default types are the standard no-level, alphanumeric and two-level shifts, along with some extensions for things like the SysRq key.

basic+pc+iso9995+norepeat Use a basic compatibility map, which allows basic level and group shifts, allow the Alt keys to be interpreted, allow group shifts in the way that ISO9995 specifies and don't repeat the return key when it is held down.

en_US(pc104)+dk+ctrl(swapcaps)+group(switch) Use an extended English/US keyboard layout for a 104-key PC keyboard, add the extra characters that you would need to use for Danish, swap the Caps Lock and Control keys, so that the control key is next to the A key and use the Right-Alt key to switch groups while the key is held down.

pc(pc104) We're using a bog-standard 104-key PC keyboard.

4 XKB Configuration Files

Working through the XKB configuration files is a bit of a chore. In most cases you don't have to. There are perfectly adequate pre-packaged configuration files already present. Before you even *think* of wading through this section, have a look at section 3.1.

However, at the end of the day, you may have to roll your own. Or you may have to understand what's in the files so that you can pick one.

The main files are in the `keycodes`, `types`, `compat`, `symbols` and `geometry` subdirectories, corresponding to the components of the same name. The `keymap`, `rules` and `semantics` subdirectories contain ways of grouping the main components together into neat bundles.

4.1 Basics

XKB configuration files are all structured in a similar way. A single configuration file contains information on a group of similar items. The file can contain a number of *variants* that contain variations on a theme.

Each variant has the following syntax: a set of options, a type and a name, followed by the variant information in braces and a semi-colon. The type describes the sort of information present, eg. `xkb_symbols` for **symbols** configuration. The name is enclosed in quotes and contains the variant name, eg. `"pc102"` — see section 3

Within each variant is actual configuration information. This information uses whatever keywords and information is appropriate to the type at hand. It is also possible to include information from other files and variants. There are two ways of including information: you can *include* or *augment*. If you include, using the syntax `include "`*`file(variant)`*`"` — note the lack of a semi-colon — then the information from the included file overrides any configuration information that already exists. If you augment, using the syntax `augment "`*`file(variant)`*`"` then the information is only added if it doesn't override something that already exists. In both cases, the *file* argument is assumed to come from the same base directory as the type, eg. **symbols** files come from the `symbols` directory.

The configuration file syntax also allows inheritance of information wherever possible. Contexts and groupings are enclosed in braces. Information from parent contexts are inherited. For example, in a **geometry** file, the top level of some geometry might have `shape.cornerRadius = 1;` to set the level of corner roundness. This piece of shape information is inherited by any defined shapes, although it can be overridden for a particular shape.

The options set the various levels of visibility and function that each variant shows. The options are summarised in table 1. The `*_keys` options are used in the **symbols** component and are used to provide hints as to what partial symbol maps contain.

4.2 Key Codes

The key codes files map a keyboard's scan codes onto useful symbolic forms. These files are the first point of contact between a keyboard and the XKB system. After that, things tend to become keyboard independent, focusing on the sorts of symbols that you want to produce and the behaviour that you want out of the keys.

The basic structure of a key codes map is something like:

```
xkb_keycodes "basic" {

    minimum= 8;
    maximum= 255;

    <TLDE> =  49;
    <AE01> =  10;
    <AE02> =  11;
...
    indicator 1 = "Caps Lock";
```

Option	Description
`default`	Marks this as the default variant in the configuration file, for use when no variant is specified. See section 3 There is a suspicion that the default entry should be the first entry in a file.
`partial`	Indicates that this bit of configuration is only partial and needs to be included into a more complete configuration.
`hidden`	A variant that can only be used within the configuration file. Used for providing useful common includes.
`alphanumeric_keys`	Indicates a partial symbol map with alphanumeric keys. See section 4.3.
`modifier_keys`	Indicates a partial symbol map with modifier keys. See section 4.3.
`keypad_keys`	Indicates a partial symbol map with keypad keys. See section 4.3.
`function_keys`	Indicates a partial symbol map with function keys. See section 4.3.
`alternate_group`	Indicates a partial symbol map with keys for an alternate group. See section 4.3.

Table 1: Variant Options

```
    indicator 2 = "Num Lock";
...
    alias <AE00> = <TLDE>;
};
```

The `xkb_keycodes "basic"` gives the type of the component (key code map) and the variant name (*basic*).

Following that, the `minimum` and `maximum` lines give the minimum and maximum key codes that the keyboard generated. If not all the key codes are used, that's not a problem.

The lines of the form `<AE01> = 49;` map a keyboard key code (49 in this case) onto a symbolic key name (`<AE01>` in this case). These lines associate key codes with the names that will be used in components such as a **symbols** component.

The convention used is to explicitly name shift and escape-like keys, but to name ordinary keys by a positional code. Hence, `<AE01>` is actually what would be the `1/!` key on an ordinary qwerty keyboard. However, the left shift key is denoted by `<LFSH>`. This convention is, presumably, so that Dvorak-like keyboards can be sensibly specified. The keyboard codes are notionally based on a conventional qwerty keyboard and are shown in figure 2

The lines of the form `alias <AE00> = <TLDE>;` allow alternate names to be associated with the same key. In the example given, the tilde key, associated with the key code 49 in this keyboard can also be referred to as `<AE00>` as well as `<TLDE>`. This facility is useful if you want to refer, sometimes, to "the leftmost key of the top row" rather than "the tilde key".

The lines of the form `indicator 1 = "Caps Lock";` enumerate the indicator LEDs that the keyboard has. I *think* that the name of the indicator is important, rather than the number, since this is referred to in other components, such as the **compat** component. It's possible that the indicies refer to the numbering of the indicators in the keyboard's hardware. The problem with this interpretation is that it violates the *symbolic* = *concrete* formulation used in the rest of the file.

Virtual indicators are also possible. I have no idea what these do.

4.3 Symbols

The **symbols** component maps the symbolic key codes created by the **keycodes** component onto whatever symbols the user wants to produce. It's at this point that peculiarities such as national character sets, the

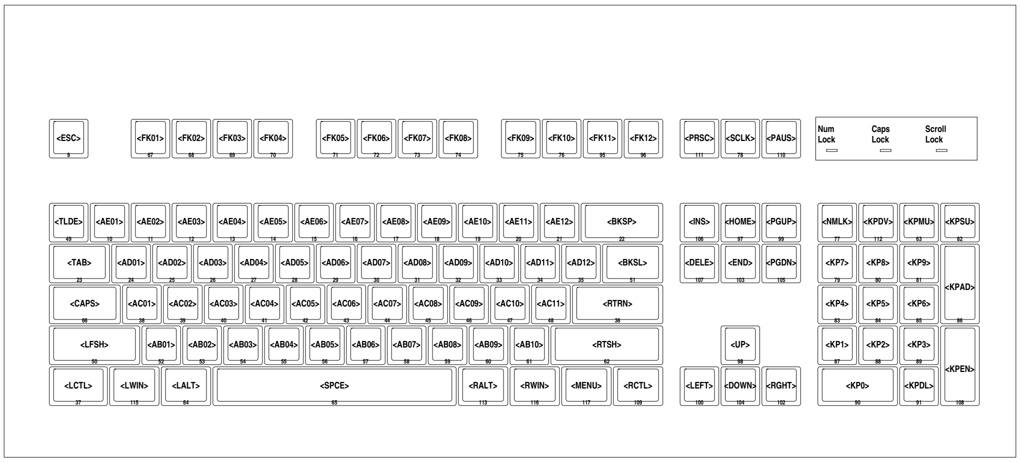

Figure 2: Key Code Naming Conventions

position of control keys and other things make an appearance. The symbols maps also handle the effect of groups and levels — see section 2.2.

Some of the symbols generated by a symbols map are not used directly, but are passed on to the **compat** component for further processing.

A symbol map file may look something like this:

```
partial alphanumeric_keys
xkb_symbols "basic" {

    name[Group1]= "US/ASCII";
    key <ESC>  {        [ Escape                        ]       };
...
    key <TLDE> {        [ quoteleft,    asciitilde      ]       };
    key <AE01> {        [         1,    exclam          ]       };
...
    modifier_map Shift  { Shift_L, Shift_R };
...
};
```

The first part of the symbol map, `partial default alphanumeric_keys xkb_symbols "basic"` simply declares that this is a symbol map named *basic*, along with a few options. The `partial` option indicates that this map doesn't cover a complete keyboard, just some interesting section of it. The `alphanumeric_keys` option describes which section of the keyboard is being covered. Multiple `*_keys` options are allowed, but if none are specified, then the map is assumed to cover a complete keyboard.

The `name[Group1]= "US/ASCII";` statement gives a name to one of the keyboard groups. Other groups can be specified by using the same syntax but with a different group name.

The line `key <TLDE> { [ quoteleft, asciitilde ] };` describes a single mapping from a key code (`<TLDE>` in this case) to a group of symbols (‘ and ˜ in this case). Symbols are named, using the symbolic names from the `/usr/X11R6/include/X11/keysymdef.h` file, as described in section 2.4. A pair of symbols enclosed in brackets indicates a pair of symbols separated by a shift level; pressing the Shift key usually shifts between two levels, but the **types** component can override this. If a single symbol is enclosed in brackets, then this symbol is used always, independent of the level.

The `key` *`code`* { `[` *`symbol`*, *`symbol`* `]` } syntax is a short form of a more general syntax that allows more flexible specification. Inside the braces, it is possible to use the syntax group[*groupname*] = [*symbol*, *symbol*] instead, with the various statements being separated by commas. This syntax is needed when extra information, such as type information is being specified for the key — see section 4.3.2.

XKB is pretty flexible with respect to modifier keys, pretty much allowing anything which looks like a modifier key to be used as one. The **types** and **compat** components handle the interpretation of various combinations of keys. However, there are many X11 programs that need to keep track of modifier key status (eg. a flight simulator) and expect input from the old modifier combinations of Lock, Shift, Control and Mod1–Mod5. The line `modifier_map Shift { Shift_L, Shift_R };` maps the left-Shift and right-Shift keys onto the old-style Shift indicator, so that older programs have some idea as to what is going on.

4.3.1 Handling Groups

Multiple groups can be specified by including other lists of symbols after the first list, with each list separated by commas. Each group needs to be named. The resulting configuration looks like:

```
name[Group1]= "US/ASCII";
name[Group2]= "Russian";
...
key <AD01> { [ q, Q ],
             [ Cyrillic_shorti, Cyrillic_SHORTI ] };
```

In this case, two groups are defined. For the Q key, two lists of symbols are given, one for the US/ASCII group and one for the Russian group.

It is also possible to specify multiple groups using the long-form syntax. For example:

```
key <AE11> {
    symbols[Group1]= [ minus, underscore ],
    symbols[Group2]= [ minus, questiondown ]
};
```

It is also possible to define a partial symbol map that only provides mappings for some higher, non-default group. In this case, the symbol mappings for the lower-level groups are empty. These sort of symbol maps can be included in with more basic maps, to allow a For example, the following defines a group 3 (only) mapping:

```
key <AD01> { [], [], [ q, Q ] };
```

4.3.2 Handling Levels

The **types** component specifies how differing levels are to be handled for various keys. By default, it would appear to be the case that most keys are automatically given a two-level or alphabetic type (see section 4.4).

In some cases, however, it may become necessary to explicitly associate level-behaviour with a key. In this case, the type and symbol map needs to be given explicitly, for example:

```
key <PRSC> {
    type= "PC_SYSRQ",
    symbols[Group1]= [ Print, Sys_Req ]
};
```

In this case, the `<PRSC>` key code is given an explicit type (`PC_SYSRQ`). The group 1 symbols are then explicitly given, using the long syntax.

4.3.3 Handling Modifiers

Most modifiers are simply specified by including the suitable symbol names; modifier keys have names, the same as any normal key. For example, the left-Control key is called `Control_L`. In most cases, a separation is maintained between left- and right-hand versions of the keys, even if you want them to do the same thing.

The **compat** component expects certain *virtual modifiers* to be defined, so that certain logical behaviours (eg. how to switch groups) can be neatly defined — see section 4.5. If a modifier key contributes to a virtual modifier, then that needs to be specified, using a `virtualMods = name` statement in a key definition. The long-form syntax needs to be used, for example:

```
key <RALT>  {
    symbols[Group1]= [ Mode_switch, Multi_key  ],
    virtualMods= AltGr
};
```

4.3.4 Control Keys

Generally, holding the Control key down and typing an alphabetic key is expected to produce a low-value control-key character, eg. Control-H produces ASCII 8. Which keys produce which control characters is, essentially, hard-wired and specified in appendix A of the XKB protocol.[3].

Which keys are control keys is specified by using the `modifier_map` statement to map keys onto the Control modifier. For example:

```
modifier_map Control { Control_L };
```

4.3.5 Special Characters

There are a huge array of "symbols" that are intended, instead, to be used to control various parts of XKB or X11. The most obvious examples are the modifier keys, discussed in section 4.3.3. In addition, there are lots of symbols designed to do things like launch a web browser or be combined with other symbols. All of these symbols have character codes.

Dead Keys These keys are intended to represent accents which are combined with other symbols to form an accented symbol. As an example, the keyname' key and the a key can be combined to form the **á** symbol. When a dead key is pressed, nothing immediately appears. Instead, XKB waits for the next character and attempts to combine it with the accent. Dead keys have names like `dead_acute`.

ISO Keys The ISO keys provide a grab-bag of features: keys for shifting group and level, keys for setting word-processor like features (such as margins) and keys for moving about the screen. All of these keys have an `ISO_` prefix.

X11 Control Keys These keys include control keys to terminate the X server and keys to flip between virtual screens.

XKB Control Keys XKB provides a number of facilities for ease of use: options to handle slow typing, key repeat, sticky keys and using keys to move the mouse pointer. See section 4 of the XKB protocol document for more information on these keys.[3]

In addition, there are pseudo-keys such as Mode_shift that are not intended to actually produce symbols. These keys are usually consumed by the **compat** component to produce actions.

Composition Keys The **Multi_key** key can be used to build accented characters and the like. Holding down the **Multi_key** key, followed by the accent character, followed by the letter will produce the accented character.[5]. The file `/usr/X11R6/lib/X11/locale/localename/Compose` contains the list of compose sequences for a particular locale or encoding (eg. iso8859-2) and what they produce. (There are other composition keys, such as **Codeinput**, but I don't know what they do.

4.4 Types

Types provide information as to the levels available for various keys and how to shift between the levels. Each key can have its own type and have differing numbers of levels and differing ways of switching between levels; eg. only alphabetic characters are shifted when the Caps Lock key is on.

The type that a key has can either be explicitly given in the **symbols** component — see section 4.3.2 — or assigned automatically. If a type is not explicitly specified, then the following rules apply:

- If there is only one symbol listed for the key, then the type is ONE_LEVEL. There are no level changes.
- If the level 1 symbol is a lower-case letter and the level 2 symbol is an upper-case letter, then the type is ALPHABETIC. Either the Shift or Caps Lock key changes levels.
- If there is a keypad symbol on any level, then the type is KEYPAD. Either the Shift or Num Lock key changes levels.
- Otherwise, the type is TWO_LEVEL. The Shift key changes levels.

An example **types** component is shown below:

```
partial default xkb_types "default" {
    virtual_modifiers LevelThree;

    type "THREE_LEVEL" {
        modifiers = Shift+LevelThree;
        map[None] = Level1;
        map[Shift] = Level2;
        map[LevelThree] = Level3;
        map[Shift+LevelThree] = Level3;
        level_name[Level1] = "Base";
        level_name[Level2] = "Shift";
        level_name[Level3] = "Level3";
    };
};
```

The `virtual_modifiers LevelThree;` line describes the virtual modifiers that are to be used. These are modifiers that have an equivalent `virtualMods = LevelThree` somewhere in the **symbols** component, or a `virtualModifier = LevelThree;` somewhere in the **compat** component. The standard shift modifiers do not need to be specified.

The `modifiers = Shift+LevelThree;` line gives the set of modifiers that need to be considered for this particular type.

The `map[Shift] = Level2;` line indicates that level 2 is to be used when the shift key is down.[4]

The lines such as `level_name[Level2] = "Shift";` give suitable names to each level. These lines are useful when you wish to do things like print out key maps. Otherwise, they have no direct effect on anything.

4.5 Compatibility Maps

The **compat** component is a shortening of "compatibility map". This seems like a rather odd name for the component that is largely concerned with translating certain key combinations into *actions*, rather than symbols. But there you are.

[4] The business of locking keys such as the Caps Lock key is handled by the **compat** component — see section 4.5. It's enough for a type to know that the key is active.

Action	Description
NoAction	Do nothing.
SetMods	Set modifier state, while the keys are held down.
LatchMods	Latch modifier state, until the next key is pressed.
LockMods	Lock modifier state, until the keys are pressed again.
SetGroup	Set current group, while the keys are held down.
LatchGroup	Latch current group, until the next key is pressed.
LockGroup	Lock current group, until the keys are pressed again.
MovePtr	Move the mouse pointer.
PointerButton	Also PtrBtn. Simulate a mouse button press.
LockPointerButton	Also LockPtrBtn and LockPtrButton. Simulate a mouse button press, locked until this key is pressed again.
SetPointerDefault	Also SetPtrDflt. Set the default select button?
ISOLock	Convert ordinary modifier key actions into lock actions while this action is active.
TerminateServer	Also Terminate. Shut down the X server.
SwitchScreen	Switch virtual X screen.
SetControls	Set the standard controls, such as slow keys or audible bell. See section 4 of the XKB protocol.[3]
LockControls	Lock the standatd controls.
MessageAction	Also ActionMessage and Message. Generate an arbitrary special-purpose XKB event.
RedirectKey	Also Redirect. Emulate the pressing of a key with a different scan code.
DeviceButton	Also DeviceBtn, DevButton and DevBtn. Emulate an event from an arbitrary input device such as a joystick.
LockDeviceButton	Also LockDeviceBtn, LockDevButton and LockDevBtn. Emulate an event from an arbitrary input device such as a joystick.
DeviceValuator	Also DeviceVal, DevValuator and DevVal. Not implemented.
Private	Generate an arbitrary event with a type and data.

Table 2: Actions in compatibility maps

Compatibility maps intercept certain combinations of keys, usually modifier keys of various sorts. These keys are translated into various actions, ranging from changing the internal state of XKB (eg. selecting the current group) to moving the mouse pointer. Usually, these key strokes are consumed by the compatibility map and disappear — although they can be passed on to other components or into the outside world.

Compatibility maps also control the various indicator lights that are displayed.

The basic structure of a compatibility map is a set of *interpret* statements that map combinations of keys onto *action* statements. The action statements usually consist of some kind of verb and a set of fields to act upon. The actions are listed in table 2 and the fields in table 3. Not every field is used in each action, table 4 lists the fields available for each action.

A major part of many compatibility maps is the handling of modifier keys. These keys need to be translated into concrete actions — group and level shifts, etc. — as well as locked or latched. A normal (unlocked, unlatched) modifier key has an effect only while it is being held down. A *locked* modifier key is on until another key press releases it. A *latched* modifier key is on until another key is pressed, at which point the modifier is released.

Field	Type	Description
clearLocks	boolean	Clear any locked modifiers when the key is released.
latchToLock	boolean	Lock a modifier when the key is released.
generateKeyEvent	boolean	Also genKeyEvent. Generate a key event as well as an action.
report	key event mask	Send this message on press or release of a key.
default		Not used.
affect	modifier group	Alter locking behaviour in which modifiers.
increment		Not used.
modifiers	modifier list	The set of modifiers to set, latch, lock or otherwise manipulate.
group	1–8	The group to set, latch, lock or otherwise manipulate.
x	integer	The mouse pointer position.
y	integer	The mouse pointer position.
accelerate	boolean	Also accel and repeat. Allow accelerated mouse movement.
button	1–5	The mouse button name or number.
value	1–5	Appears to be a synonym of button.
controls	global control name	Also ctrls. The name of one of the various global controls such as slow keys or overlay.
type	0–255	The internal type of a private action.
count	0–255	The number of button presses for something like a double-click.
screen	0–255	Virtual screen number
sameServer	boolean	Also same. Switch to a server screen, rather than a console screen.
data	String	Data for a private action or message action. Instead of a string, data can also be an arbitrary numeric array, set with `data[0] = 113`
device	1–255	Also dev. The device number for non-mouse input devices.
keyCode	key name	Also key and kc. The key code to emulate.
clearModifiers	modifiers	Also clearMods. Modifiers to remove from the current modifier set.

Table 3: Fields in compatibility maps

Action	Fields
NoAction	
SetMods	clearLocks, latchToLock, modifiers
LatchMods	clearLocks, latchToLock, modifiers
LockMods	modifiers
SetGroup	clearLocks, latchToLock, group
LatchGroup	clearLocks, latchToLock, group
LockGroup	group
MovePtr	x, y, accelerate
PointerButton	button, count
LockPointerButton	button, count
SetPointerDefault	affect, button
ISOLock	modifiers, group, affect
TerminateServer	
SwitchScreen	screen, sameServer
SetControls	controls
LockControls	controls
MessageAction	report, generateKeyEvent, data
RedirectKey	keycode, modsToClear, modifiers
DeviceButton	button, affect, count, device
LockDeviceButton	button, affect, count, device
Private	type, data

Table 4: Fields for actions

A compatibility map looks something like:

```
default xkb_compatibility "basic"  {
    virtual_modifiers NumLock,AltGr;
...
    interpret.repeat= False;
    setMods.clearLocks= True;
...
    interpret Shift_Lock+AnyOf(Shift+Lock) {
        action= LockMods(modifiers=Shift);
    };
...
    group 2 = AltGr;
...
    indicator.allowExplicit= False;
...
    indicator "Caps Lock" {
        whichModState= Locked;
        modifiers= Lock;
    };
...
};
```

The `default xkb_compatibility "basic"` line gives the usual option, type and name information, same as any other component.

The `virtual_modifiers NumLock,AltGr;` line lists any virtual modifiers that might be set or examined

The `interpret.repeat= False;` and `setMods.clearLocks= True;` lines set one of the default fields for part of the compatibility maps. Both actions (eg. `setMods`) and certain syntax (eg. `interpret`) can have defaults set.

The line `interpret Shift_Lock+AnyOf(Shift+Lock)` tells the compatibility map what keys trigger actions. The first part (Shift_Lock) is the name of the key that has been pressed. The second part gives the modifier keys that need to be set to trigger this action. In this case, either the Shift or Lock modifier needs to be set. Specifying modifier keys is optional, or specified as `Any`. If more than one key is possible, as above, then `AnyOf`, `AllOf`, `Exactly` or `AnyOfOrNone` are possible combinations of modifiers.

The line `action= LockMods(modifiers=Shift);` gives the action that the keys in the `interpret` line are supposed to produce. In this example, `LockMods` is the action (see table 2) and `modifiers` is the field (see table 3) to be set. Multiple fields can be separated by commas. The right hand side of a field assignment can be an expression with the usual arithmetic operators. For numeric fields, instead of straight = assignment, a signed value indicates a relative change — eg. `group= +1` increments the current group.

The line `group 2 = AltGr;` maps a group value onto a modifier state. This is for the benefit of older programs that know nothing of the wonders of XKB. In this case, being in group 2 means that the AltGr modifier is set.

The `indicator "Caps Lock"` line declares the conditions under which an indicator is lit. The name of the indicator ("Caps Lock") in this case is mapped onto the indicator names given in the **keycodes** component — see section 2.3.

The line `whichModState= Locked;` gives the state that the particular modifier needs to be in to activate this indicator. Possible states are Base, Latched, Locked, Effective, Any or None.

The line `modifiers= Lock;` gives the list of modifiers that need to be in the correct state to activate this indicator.

4.6 Geometry

The **geometry** component is probably the most useless part of XKB. So we're going to do it last. Essentially, this component supplies information on the physical layout of certain keyboards, so that programs such as **xkbprint** can produce sensible looking output.

Naturally, for something so trivial, the **geometry** files are some of the most complex. A sample file is shown below:

```
default xkb_geometry "pc101" {

    description= "Generic 101";
    width= 470;
    height= 210;
...
    shape.cornerRadius= 1;
...
    shape "NORM" { { [ 18,18] }, { [2,1], [ 16,16] } };
...
    solid "LedPanel" {
        shape= "LEDS";
        top=  52;
        left= 377;
        color= "grey10";
    };
...
    indicator "NumLock"     { left= 382; };
...
    text "NumLockLabel"     { left= 378; text="Num Lock"; };
...
    section "Function" {
        top= 52;
```

```
        row {
            top= 1;
            keys {  { <ESC>, color="grey20" },
                    { <FK01>, 20 }, <FK02>, <FK03>, <FK04>,
...
    alias <AC00> = <CAPS>;
...
};
```

The `default xkb_geometry "pc101"` line contains the usual options, type and name declaration. The `description= "Generic 101";` line gives a descriptive name to the keyboard.

The line `width= 470;` gives the total width of the keyboard. All lengths in **geometry** declarations are multiples of 1mm, so 470 is 47cm.[5]

The line `shape.cornerRadius= 1;` defines a default setting for a shape field. In this case, the corner radius of a join is set to 1mm. Other items, such as solids, have other fields, these are listed below.

The `shape "NORM" { { [ 18,18] }, { [2,1], [ 16,16] } };` line declares a *shape*. A shape is a named drawing outline that can be used elsewhere to draw something like a key or an indicator. This shape defines the look of a normal key. All coordinates are from the upper-left corner and increase rightwards and downwards. The origin of a shape is always $(0,0)$ in the upper-left corner; this position is shifted to wherever a shape needs to be drawn. A shape consists of a list of outlines, with each outline representing a closed figure. An outline is a list of coordinates in $[x,y]$ form, each list enclosed in braces. The interpretation of an outline depends on the number of coordinates in the list: if one coordinate is given, then a box is drawn from $(0,0)$ to the coordinate; if two coordinates are given, then a box is drawn from the first coordinate to the second; if three or more coordinates are given, then an arbitrary closed figure is drawn, with a vertex at each coordinate. In the example above, a box is first drawn from $(0,0)$ to $(18,18)$ and then another box from $(2,1)$ to $(16,16)$ to give a suitable-looking key outline.

The section beginning with `solid "LedPanel"` draws a solid area of colour. A *solid* is an example of a *doodad*; a piece of decoration designed to fill out the appearance of the keyboard. Other doodads are indicators, outlines, text and logos. The fields of a solid are `left` and `top`, which give the start position of the solid, `shape`, which gives a named shape for the solid to draw and `color`, which gives the colour of the solid. Colours can be any of the named X11 colours. The field `priority` presumably gives the drawing order for overlaid items.

The `indicator "NumLock" { left= 382; };` line declares an *indicator* doodad, suitably named. The possible fields for an indicator are: `onColor`, `offColor`, `left`, `top`, `priority` and `shape`. I'm not entirely sure how an indicator is mapped onto an appropriate logical indicator from the **compat** component or the **keycodes** component. It may be related to the numbering of the indicators.

The `text "NumLockLabel" { left= 378; text="Num Lock"; };` line provides a named *text* doodad: a piece of text to be placed somewhere. The possible fields for a text doodad are: `width`, `angle`, `height`, `text`, `top`, `left`, `font`, `slant`, `weight`, `fontwidth`, `variant`, `encoding`, `xfontname`, `fontsize`, `priority` and `color`. Most of these fields are concerned with the X11 font system. Most fields will assume sensible default values if left alone.

The `section "Function"` line starts the definition of a keyboard *section*: a block of keys with similar functions. Examples of sections are the function keys, the alphanumeric keyboard and the numeric keypad. Sections have the following fields: `priority`, `top`, `left`, `width`, `height` and `angle`. Most of these will assume sensible defaults if left out. Sections are composed of a number of rows.

The `row` line starts a *row*: a row of keys. Rows have the following fields: `top`, `left` and `vertical`. The coordinates of a row are relative to the coordinates of the enclosing section. Each row consists of a list of keys.

The `keys` line defines the list of keys in a row. Each key is listed in turn, giving the keycode of the key. For example, `<FK02>` is the F2 key. Each key has the following fields: `gap` the spacing from

[5] Yes, the XKB protocol specification says 0.1mm. Looking at actual geometry files, though, reveals this to be rubbish, unless people really use 4.7cm wide keyboards and I'm in a strange dimensional vortex. Since the results scale in Postscript to fit a piece of paper, I suspect that it doesn't matter.

the previous key, `shape` the named shape of the key, `color` the color of the key and `name` the symbolic key code. Normally, these fields are set to defaults. However, if a field needs to be set, the syntax { `<ESC>, color="grey20"` } is used. Usually, the fields that need to be set are the gap and shape fields. The syntax { `<FK01>, 20` } is a short form that sets the keycode and the gap. The syntax { `<SPCE>, "SPCE"` } is a short form that sets the keycode and the shape.

4.7 Directory Files

To allow rapid lookup of components, XKB keeps *directory* files in the root XKB configuration directory. These files have a name of the form *component*`.dir`. For example, the **compat** directory file is called `compat.dir`.

Each possible component and variant has a line in the directory file. A typical line is as follows:

```
-dp----- a------- macintosh/se(basic)
```

The `macintosh/se(basic)` gives the component name. The two fields in front give various interesting option flags — see table 1.. The first flag field can be a combination of `hdp-----` with h for hidden, d for default and p for partial. The second flag field can be a combination of `amkfg--` with a for alphanumeric, m for modifier, k for keypad, f for function keys and g for alternate group; although their use seems to be somewhat spotty.

4.8 Grouping Components

As well as the main five components, there are a few grouping components designed to allow easier specification of XKB configurations.

4.8.1 Semantics

The **semantics** component provides named combinations of the **types** and **compat** components. This component doesn't seem to be used very much.

4.8.2 Key Maps

The **keymap** component provides named mappings for all the main components, allowing common keyboard configurations to be given simple names. An example keymap is:

```
xkb_keymap "en_US" {
    xkb_keycodes        { include "xfree86"             };
    xkb_types           { include "default"             };
    xkb_compatibility   { include "default"             };
    xkb_symbols         { include "en_US(pc105)"        };
    xkb_geometry        { include "pc"                  };
};
```

This keymap defines the `en_US` keymap. Essentially, all that is happening here is that a collection of appropriate components have been rolled up and named "en_US".

Interestingly, the use of "include"s, rather than simple names suggests that the components could be extended within the keymap itself, presumably using the same syntax as an ordinary component. No examples of this exist, however.

4.8.3 Rules

The key maps described in section 4.8.2 are pretty much what an end-user would like to see for XKB configuration. The more technical aspects of the components are hidden and you can select from a few national variants. For those with PC keyboards, this is largely all that is needed.

For those with more specialised needs or unusual keyboards, however, the simplicity gets in the way of using the **keymap** component. The **rules** component provides a more flexible approach, allowing a user-centred way of describing keyboard layouts while also allowing a more mix-and-match way of choosing components. A list of parameters are given, such as "model" and "layout". These parameters are matched against a set of rules until a match is found and the corresponding components chosen.

A sample rules file looks like:

```
! model         =       keycodes        geometry
  pc101         =       xfree86         pc(pc101)
...
! model         layout  =       symbols
  pc102         intl    =       us(pc102compose)
  pc104         *       =       en_US(pc104)+%l%(v)
  *             *       =       en_US(pc101)+%l%(v)
...
! option                =       symbols
  grp:switch            =       +group(switch)
...
```

The lines of the form `! model = keycodes geometry` give the mapping from the rules parameters on the left (`model` in this case) to the components on the right (**keycodes** and **geometry** in this case). Multiple rules parameters and multiple components are all possible.

The lines of the form `pc101 = xfree86 pc(pc101)` give mappings from the declared rules parameters to the declared components. In this case, the keyboard model `pc101` is mapped onto the default `xfree86` keycodes and the `pc101` variant of the `pc` geometry. Lines which have a `*` in a rules parameter's spot match any value of the parameter.

Some lines contain parameters of the form `+%l%(v)`. These lines substitute the layout parameter (%l) and the variant parameter (%(v)) to build a suitable inclusion. For example, if the configuration had the layout set to `ru` and the variant set to `winkeys` then the line `en_US(pc104)+%l%(v)` becomes `en_US(pc104)+ru(winkeys)`. It would appear that the % causes the first alphabetic character after the sign to be expanded. These parameter forms are used to build sensibly parameterised national variants, since the national variants have no symbols for things like the keypad keys.

A line of the form `grp:switch = +group(switch)` indicates something that will be added to an existing component definition. These lines are usually used for options that need to be added to a basic **symbols** specification.

In the **rules** directory, there are also files with a `.lst` extension. These files contain descriptions of the various rules parameters and the values they may have. A suitable UI could use these values to be a little more user-friendly when displaying values.

5 XKB Programs

There are a number of programs which come with XKB for management and debugging purposes. These all have perfectly adequate **man** pages, or have a sensible `-help` option. I'll just mention a few things that might be useful here.

setxkbmap This program allows an XKB map to be installed. Components can be specified either directly, with arguments such as `-symbols en_US(pc104)+ru` or via the **rules** parameters, listed in section 4.8.3.

xkbcomp Compile an XKB keyboard description. This program converts the contents of a suitably specified set of configuration files into a form suitable for an X-server to use.

What is also interesting about this program is that it can also be used, using the `-xkb` option, to produce a source file for the current XKB configuration. For example, command `xkbcomp :0.0 -xkb` will produce a file called `server-0_0.xkb` which contains the complete configuration source for server 0.0.

Figure 3: The Happy Hacking Keyboard Lite

Switch			L-Alt		L-◇		R-◇		R-Alt	
2	3	4	Key	Code	Key	Code	Key	Code	Key	Code
0	0	0	L-Alt	64	Muhenken	131	Henkan	129	R-Alt	113
0	0	1	Muhenken	131	L-Alt	64	R-Alt	113	Henkan	129
0	1	0	L-Alt	64	Fn	-	Henkan	129	R-Alt	113
0	1	1	Fn	-	L-Alt	64	R-Alt	113	Henkan	129
1	0	0	L-Alt	64	L-Windows	115	R-Windows	116	R-Alt	113
1	0	1	L-Windows	115	L-Alt	64	R-Alt	113	R-Windows	116
1	1	0	L-Alt	64	Fn	-	R-Windows	116	R-Alt	113
1	1	1	Fn	-	L-Alt	64	R-Alt	113	R-Windows	116

Table 5: Diamond Key Options on the Happy hacking Keyboard

xkbprint Produce a graphical map of the keyboard. This is what the **geometry** component is used for. The map is usually a Postscript map of the keyboard, showing the keys for each character. A command such as `xkbprint :0.0` will produce a file called `server-0_0.ps` with a map of the basic keyboard. Options such as `-label` *`type`* allow things such as the scan codes or key names to be printed, instead.

A An Example Keyboard: The Happy Hacking Keyboard Lite

This section is the point, for me at least, of writing all the above. All I wanted to do was to get my keyboard working and producing useful symbols. Figure 3 is a picture of the HHK. As you can see, this is a pretty stripped-down keyboard, with no function or editing keys or keypad. Some of these keys are available in combination with a special Fn key.

The two keys with a diamond on them, either side of the space key can be set to produce a number of key codes. The possible combinations are shown in table 5. Two keys, Muhenken and Henken are used on Japanese keyboards to access extra groups. The other keys are the Alt and Windows keys familiar from normal PC keyboards.

Setting up for the HHK largely involves choosing what bits of already existing XKB configuration can be pressed into service. The full set of configuration files can be found in `http://www.charvolant.org/~doug/xkb/hhk.tar.gz`.

A.1 Geometry

We might as well start with the physical shape of the keyboard. Since there is nothing even close to a suitable geometry file for the HHK, we'll need to roll our own.

A complication arises when the effects of the switches described in table 5 are considered. The solution I've adopted is to split the keyboard into two sections, an *Alphanumeric* section that covers most of the keyboard and a *Space* section that covers the row containing the alt, diamond and spacebar keys. Those switch settings that give a windows-like keyboard (switch 2 on) are labelled `win1-4`. Those switch settings that give a Japanese keyboard (switch 2 off) are labelled `jp1-4`. The default setting is `win1`.

Another complication is the Fn key, which has a position on the keyboard, but does not generate a scan code. I've used the symbolic name `<FN>` for this key, which doesn't seem to be used anywhere else.

The resulting file looks like this:

```
// SW = 100 Default layout with windows keys
default xkb_geometry "win1" {
    include "hhk(basic)"

    key.gap = 1;
    section "Space" {
        top = 86;
        left = 30;
        row {
            top = 1;
            keys {
                <LALT>,
                { <LWIN>, "DIAM" },
                { <SPCE>, "SPCE" },
                { <RWIN>, "DIAM" },
                <RALT>
            };
        };
    };
};
...
// Basic layout for non-switchable keys
partial hidden xkb_geometry "basic" {
...
    shape "NORM" { { [18, 18] }, { [2, 1], [14, 14] } };
    shape "DIAM" { { [28, 18] }, { [2, 1], [24, 14] } };
...
    text "PFULabel" { left = 16; top = 2; fontsize = 20; text = "PFU"; };
...
    section "Alphanumeric" {
...
        row {
            top = 20;
            keys {
                { <TAB>, "TABK" },
                <AD01>, <AD02>, <AD03>, <AD04>, <AD05>,
```

Figure 4 shows the resulting geometry.

Once we have this file, we can install it in the `/usr/X11R6/lib/X11/xkb/geometry` directory as `hhk`. We then need to add the following lines to `/usr/X11R6/lib/X11/xkb/geometry.dir`:

```
-d------ -------- hhk(win1)
```

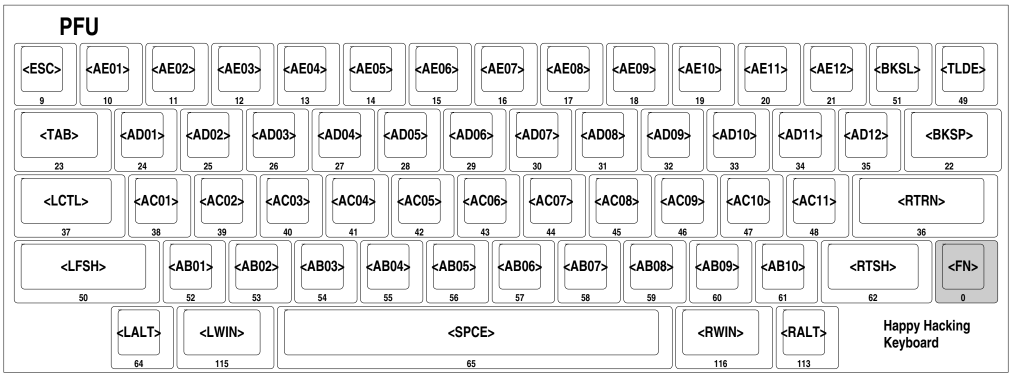

Figure 4: Happy Hacking Keyboard Geometry

```
-------- -------- hhk(win2)
-------- -------- hhk(win3)
-------- -------- hhk(win4)
-------- -------- hhk(jp1)
-------- -------- hhk(jp2)
-------- -------- hhk(jp3)
-------- -------- hhk(jp4)
```

A.2 Key Codes

The HHK acts as an ordinary — just small and perfectly formed — PC keyboard. As a result the `xfree86` key codes can be used without modification.

The only key not present in the `xfree86` key codes is the Fn key. However, this key does not generate a scan code and can be ignored.

The other complication is that the HHK has no numeric keypad. For the most part, this is not a problem. The only difficulty arises when the keypad keys are used for such complications as mouse movements.

A.3 Symbols

The symbols present shouldn't cause any difficulty, since they *should* be keyboard independent. I want to use the various useful characters provided by the ISO9995-3 symbol map. However, the `en_US(104)` symbol map already thoughtfully provides these, with the R-Alt key as the group-shift key and the R-Windows key as the compose key.

To get a map of these symbols, use the command `xkbprint -color -label symbols -lg 2 :0.0`

A.4 Types and Compatibility Maps

The standard `complete` set of types works perfectly well with this keyboard. As does the standard `complete` set of compatibility maps. The only loss here is that the keypad can't be used for moving the mouse pointer; unfortunate, but hardly fatal.

A.5 Rules

Once all the appropriate files have been added, the **rules** files need to be changed to allow this keyboard to be used. To do this, we need to add another keyboard model: `hhk`. Once we have done this, we can add lines to the rules file that allow suitable **geometry** and other components to be added. The lines which need to be added are:

```
! model         =       keycodes        geometry
  hhk           =       xfree86         hhk
...
! model         layout  =       symbols
  hhk           us      =       us(pc104)
  hhk           en_US   =       en_US(pc104)
  hhk           *       =       en_US(pc104)+%l(%v)
```

The first line maps the HHK onto suitable keycodes and geometry. The other lines provide symbol maps for bog-standard US, US with all the interesting extra ISO9995-3 keys and for other national variants.

References

[1] *A^+*.
http://www.aplusdev.org/.

[2] Nikolai Bezroukov. A second look at the cathedral and bazaar. *First Monday*, 4(12), December 1999.
http://firstmonday.org/issues/issue4_12/bezroukov/index.html.

[3] Erik Fortune. *The X Keyboard Extension: Protocol Specification*. X Consortium, 1996.
http://www.x-docs.org/XKB/XKBproto.pdf.

[4] *Happy Hacking Keyboard*.
http://www.pfuca.com/products/hhkb/hhkbindex.html.

[5] *Tutorial: Typing accented characters in X-windows*.
http://www.sober.com/content/accented_characters.html.

[6] Ivan Pascal. *X Keyboard Extension*.
http://www.tsu.ru/~pascal/en/xkb/.

[7] Robert W. Scheifler. *X Windows System Protocol*. X Consortium, 1994.
http://www.x-docs.org/XProtocol/proto.pdf.

[8] *XFree86*.
http://www.xfree86.org.