Keyboard configuration for Unicode input on Linux

Jean-Yves Moyen

July 17, 2017

Abstract

This document explains how to configure your keyboard (XKB + XCompose) to input Unicode characters. Rather than lengthy XCompose sequences, I prefer to use dead keys for short sequences and to use some of the unused keys (like the Window or Menu keys) as extra modifiers to access these. The proposed solution is purely a user-based configuration and does not require privileged access.

This document is probably way more verbose than needed. I know.

1 Goals and aims

1.1 What I want

I am only writing text in Western European languages, hence the usual keyboard configurations are normally good for me. I do, however, insert "special" characters quite often, most likely mathematical characters in formulas. Thus, I'd like to have a quick and easy way to make use of Unicode and access these characters.

Because my typical use-case is to insert special characters one by one, I do not need to totally switch my keyboard layout. Typically, people writing both in English and Japanese do have a way to switch the keyboard between layouts because they alternately need to write long sequences of characters in both scripts. In my case, the special characters are rare and usually lonely, so I don't need a new layout.

The main mechanism existing for inputing special character is XCompose. It is very easy to find online examples of `.XCompose` files with long lists of sequences for special characters. This is, indeed, very convenient. The main drawback of these files is the sometimes rather long sequences that need to be typed for a single character. For example, something like :

`<Multi_key> <d> <d> <a> <g> : "‡" U2021 # DOUBLE DAGGER`

I do find it rather inconvenient to type a sequence of 5 keys to enter a single character... Plus, as a LaTeX user, this has little advantage over typing `\ddag`.

A different (well, actually not) mechanism is the use of *dead keys*. A dead key produce nothing when pressed but wait for the next character and combines with it. Dead keys are typically used for producing diacritics. Typically, some layout have `dead_acute`, `dead_caron` or other dead diacritics accessible. This mechanism is not limited to diacratics. Indeed, it is possible to have a `dead_greek` key on your keyboard. Pressing it will turn the next character into its Greek version (or the closer think there is). On the bépo layout, `dead_greek` is assigned to AltGr+g, making it easy to type and inputing greek characters in texts.

I find it more convenient to use a dead keys mechanism as it requires less keystrokes to produce the special characters. Thus, I want, typically:

- to have a `dead_blackboard_bold` accessible on my keyboard;
- to tell the system that `dead_blackboard_bold` should turn the next character into its "Blackboard bold" version (*e.g.* `dead_blackboard_bold` followed by 'N' should become $\mathbb{N}$);
- to assign these dead keys to relatively easily accessible keys. Because I want room for a lot of special characters (better safe than sorry), and because AltGr+key is already assigned for most keys on my keyboard, I'd like to use some other keys as modifiers. Because I'm not really using the Window and Menu keys, they are perfect candidates for this.

1.2 What this is not about

I will only explain how to create dead keys to input special characters. I will not explain how to install the correct fonts for your computer to display these characters. I will not explain how to tell LaTeX(or rather its more modern versions) to handle these characters. All this is another subject and other guides are better suited than this one to explain it.

While I will enter a bit into XKB configuration, I do not intend to write an XKB guide. Better guides exist. Moreover, my handling of XKB is very dirty and a proper layout should be built to make this work cleanly.

1.3 Warning

These instructions come with **no warranty**. Use at your own risk.

Changing your XKB configuration can result in a totally "broken" keyboard. For example, one can make the `Enter` key totally inactive. The best way to solve these troubles it to not make the change permanent (*i.e.* not put them in your configuration files) until they are properly tested. That way, you can go back to a working keyboard by rebooting your computer...

Changing your XCompose configuration can also result in bad stuff. Probably not as bad as XKB, but still enough to make the keyboard unusable (*e.g.* unable to produce the character 'a' anymore). Here also, exercise caution when making changes.

Your Window Manager might already use some of the keys for its own shortcuts (*e.g.* Windows+d for "Show Desktop"). This can result in strange effects while making tests.

Using xmodmap together with heavy changes to XKB might be a bad idea. At the same time, if you start making changes to XKB, you can do whatever xmodmap was doing but better. This will obviously requires a bit more changes than what is described here.

XKB configuration can be changed on the fly and changes will take effect as soon as you load the new file.

XCompose configuration is only read at the start of an application. Thus, you need to launch a new application to make tests. Some applications actually clone themselves rather than launching a new instance (typically, some terminal do that, and emacs may do it depending on your configuration). Be sure to find an application where you can easily launch new instances when making XCompose tests.

If you are using a Typematrix keyboard, the two keys "Shuffle" and "Documents" are actually programmed at hardware level. The system cannot tell the difference between pressing "Shuffle" and "Alt+Tab". You can change this by switching to 106 mode. This is a physical change on the keyboard and probably also requires you to change your logical keyboard configuration from the 102 mode to the 106 mode in order to tell the system how to handle these keys (`dpkg-reconfigure keyboard-configuration` on Debian, this does require root access). Do that before starting to make other changes. Doing it also remap 4 other keys. If you happened to use them, you'll need to manually undo these changes (more on that later).

2 What exists and what doesn't exist

2.1 Pressing a key

When you press a key, the keyboard sends a *keyscan* to the computer. These keyscans are somewhat standard (I think). Next, some low level part of the computer (kernel? driver?) turns the keyscan into a numerical *keycode*. This is where we can start doing stuff.

Keycodes are somewhat standard (I think). For normal keys, they correspond to the physical position of the key. For special keys (Enter, Shift, ...) they are special keycodes for these keys.

Next, XKB get the numerical keycodes and turn them first into *keylabels* (again, positional for standard keys, special names for special keys) and then into *keysyms*. The keylables are very standard names. The keysyms are mostly regular characters. The keylabel to keysym translation depends on your keyboard layout. Typically, the top left alphabetical key (keylabel `AB01`) is mapped onto the keysym 'q' for an English (qwerty) layout; 'a' for a French (azerty) layout; and 'b' for a bépo layout.

XKB also handle *modifiers*. That is, pressing the 'a' key alone produces the 'a' keysym, but pressing it together with the Shift key produces the 'A' keysym. Each key may accept up to 8 modifiers (hence up to

256 different keysyms) but usually only accepts 2: Shift and AltGr (called Mod5 or `ISO_Level3_Shift`); hence can generate up to 4 keysyms (normal, Shift, AltGr, Shift+AltGr).

Your trusted friend for tests here is `xev`. Here is a typical display of `xev` when pressing a key:

```
KeyPress event, serial 37, synthetic NO, window 0x4200001,
    root 0xcc, subw 0x0, time 5814898, (64,100), root:(939,692),
    state 0x80, keycode 59 (keysym 0xfe8c, dead_greek), same_screen YES,
    XLookupString gives 0 bytes:
    XmbLookupString gives 0 bytes:
    XFilterEvent returns: True
```

The important line is the third: `state 0x80, keycode 59 (keysym 0xfe8c, dead_greek)` which tells you both the keycode read (59), the keysym generated (`dead_greek`) and the list of modifiers used at the time the key was pressed (the state, `0x80` which is a bitstring, here `0b10000000`, telling that only one modifier was pressed). Pressing the key whose keycode is 59 without any modifiers results in another keysym. An intensive use of `xev` is usually needed to inspect what is happening with your keyboard configuration.

Finally, keysyms are intercepted by XCompose. XCompose reads sequences of keysyms and turn them into another symbol. The dead keys mechanism is actually handled by the system-wide XCompose file.

Only after XCompose has done its magic does the application receive the symbol typed to display (or process, or whatever).

2.2 The good, the bad and the ugly

Many people already use XCompose to generate special characters, and many example of `.XCompose` files can be found online. The usual paradigm in these files is "special characters are inputed by a sequence of keys which starts with the Compose key (keysym `Multi_key`)". As I said, I prefer to have shorter key sequences and using dead keys. Obviously, shorter sequences means potentially harder mnemonics, but a good choice of dead keys can help in many cases (*e.g.* the choice of AltGr+g for `dead_greek` on the bépo layout is easy to remember).

The good part about XCompose is that it can interpret any sequence of keysyms, not just the ones starting with `Multi_key`. Thus, it is possible for example to turn the 'b' key into a `dead_blackboard_bold` by adding rules like:

```
<b><N> : "ℕ"
```

This is , however, not a good idea as you probably don't want 'b' to become a dead key... But if we can find another key (or keysym) to use, we're good.

The good part about XCompose is that it can also handle some modifiers for keys. Thus, a better attempt would be:

```
Alt<b><N> : "ℕ"
```

which turns Alt+b into a `dead_blackboard_bold`.

The bad part about XCompose, however, is that it can only handle a handful of modifiers, namely Shift, Ctrl, Alt and Meta. While this is theoretically enough for many purposes, this is not practicable because:

- Shift+key is interpreted as a different keysym by XKB, thus XCompose never receives Shift+a but always A, making it hardly usable. And of course, you don't want to turn your uppercase letters into dead keys.
- Ctrl is similarly widely used by many applications. If Ctlr+c is turned into a dead key, copy/paste might become difficult...
- Alt and Meta are better candidates but still somewhat used by applications. Especially, emacs uses Meta and often uses Alt as a substitute for Meta (because many keyboards do not have a properly defined Meta key).

Thus, XCompose alone is not able to solve my problem and I need to go into XKB...

Note that having a proper Meta key separate from Alt might be enough to emulate a lot of dead keys with Alt+key. This, however, does not allow as many possibilities and already requires digging into XKB. Since I need to go down there, lot's do it fully.

The good part about XKB is that with up to 8 modifiers –hence up to 256 keysyms– per key, I have more than enough to do whatever I want. Theoretically, with around 50 normal keys (alphabetical, numerical, punctuation, ...) and 250 keysyms for each, we could generate around 12,000 keysyms, probably more than I need. However, this makes the mnemonics extremely hard to remember. And of course, pressing up to 9 keys (8 modifiers+key) at the same time is probably more funny to watch than to do.

But a combination of XKB and XCompose can certainly do the trick. We tell XKB to generate dead keys keysyms for certain keys and XCompose to interpret them. Because we now allow sequences of 2 keys (+modifiers), we don't need that many modifiers. My choice here is to handle two more modifiers (called Hyper and Super) and to have the regular key generate 4 new keysyms for Hyper, Shift+Hyper, Super, Shift+Super. Adding 4 keysyms per key allows around 200 new dead keys and thus around 10,000 new symbols, more if the second key is itself modified (typically normal/Shift). Moreover, I am only adding states that requires 1 or 2 modifier keys to be pressed, similar to the existing AltGr/Shift+AltGr, thus this stays physically doable.

The bad part about XKB is that it is not possible to create new keysyms. The list of existing keysyms is in a `.h` file and we don't want to recompile XKB after adding each new keysym...

The ugly trick is that XKB (and XCompose) accept any Unicode character (entered as Uxxxx) as a keysym. Thus, we are going to use Unicode characters as "pseudo keysyms". Fortunately, Unicode has some reserved areas, the *Private Use Areas*, that are reserved for private use.

The ugly part is also editing your XKB configuration to create Hyper and Super keys where you want them to be...

2.3 Roadmap

So, what we have do to is:

1. Assign Hyper and Super to some keys.
2. Make regular keys accept Hyper and Super (with or without Shift) as modifiers.
3. Turn the dirty pseudo keysyms (Unicode characters from the PUA) into nicer human readable virtual keysyms.
4. Make shareable and human readable `.XCompose` file using these virtual keysyms.

The steps are both in logical order of what happen when a key is pressed; in decreasing order of difficulty (the closer to the hardware, the messier); and in increasing order of work-intensivity. Steps 1 and 2 only need to be done once and for all; step 3 requires a bit of work for each new virtual keysym one want to use; and step 4 requires copying and adapting scores of similar lines of `.XCompose` (typically, one for each letter, lowercase and uppercase, after a new dead key is introduced).

Step 3 is not strictly necessarily, but it allows to make step 4 much more human-readable. Since step 4 is the work-intensive part, it's also the one we want to be human readable.

3 XKB

Warning! It is easy to make your keyboard unusable while modifying the XKB configuration.

Do not render the changes permanent until they are thoroughly tested. That way, you can always go back to default settings by rebooting.

BÉPO users can find a pre-made keyboard configuration file at `https://http://www.lipn.fr/~moyen/Unicode_keyboard/bepo_hyper_super.txt`. It's meant to be used on a TypeMatrix keyboard in 106 mode. Simply load it with `xkbcomp bepo_hyper_super.txt $DISPLAY`. It maps `Hyper` to both the Left Window key and the Document key; and `Super` to the Menu key. Comes with no warrany, use at your own risks.

3.1 Modifying XKB configuration

Some parts of the XKB configuration are common to most keyboards and layouts (for example, the fact that the `LSHF` keylabel should correspond to the `Shift_L` keysym, or the fact that some keys accept both Shift and AltGr as modifiers) while some other parts are layout-dependent (sometimes maybe even physical keyboard dependent), for example, the fact that the upper-left alphabetical key (keylabel `AB01`) should correspond to the keysym 'a', 'q', ...

In order to avoid duplicating common parts, XKB builds your custom configuration by assembling several pieces. The pieces can be found in the subdirectories of `/usr/share/X11/xkb/`, and there are many of them. The pieces actually used to build your keyboard can be listed with

```
$ setxkbmap -print
xkb_keymap {
xkb_keycodes  { include "evdev+aliases(azerty)" };
xkb_types     { include "complete" };
xkb_compat    { include "complete" };
xkb_symbols   { include "pc+fr(bepo)+inet(evdev)+compose(caps)+terminate(ctrl_alt_bksp)"
};
xkb_geometry  { include "typematrix(tm2030USB-106)" };
};
```

Which tells, for example, that XKB is taking the `pc` file from the `symbols` subdirectory, as well as the `bepo` part of the `fr` file in the same directory, ... Note that `dpkg-reconfigure keyboard-configuration` simply asks you which parts you want to include in your XKB configuration.

As can be expected, building a clean new layout is somewhat delicate. I'm going for a quick and dirty solution which is easier to do and also does not required priviliged access. However, it will build a custom XKB configuration with all the drawbacks it has (notably of not adapting to any update in the mainstream configuration).

The XKB configuration resulting by including all these bits can be saved with `xkbcomp $DISPLAY keyboard_old.txt`. Make a copy of this file in `keyboard_new.txt`. After changing it, the new configuration can be loaded with `xkbcomp keyboard_new.txt $DISPLAY`. The global syntax is `xkbcomp source destination` where `$DISPLAY` stands for the current configuration.

Safety first! I strongly suggest that you put in your CLI history the line `xkbcomp keyboard_old.txt $DISPLAY`. That way, if you break something while building the new configuration, you may simply go back in history to restore the original one (hopefully, you haven't also break the arrows and Enter keys...) You may also want to remove writing right on the `keyboard_old` file, or even its directory, just to be sure...

Loading a configuration will output some error messages if things are wrong (*e.g.* you used an undeclared keysym). It does happen that even the original configuration has some unimportant errors (in my case, undefined keycodes)...

Of course, once everything is fixed, you'll probably want to add the load line somewhere in your configuration files and do it automatically at every login...

3.2 What's inside

Get a look into the `keyboard_new.txt` file. This is where we're going to make changes. The file uses C-style line comments (everything after a `//` is a comment).

It should start with a `xkb_keycodes` section which contains lines such as `<AE01> = 10;`. These lines assign a keylabel (on the left) to each keycode (on the right). We're not changing that. Note that it is maybe easier to create new modifiers by changing that rather than the keylabel/keysym association, but it felt a bit too low level for me to change...

Next, you have a `xkb_types` section. Each key may accept up to 8 modifiers, but to reduce the exploration space when performing the modifiers+keylabel to keysym association, each key receive a *type* telling which modifiers are allowed with this key. The section should start with the `ONE_LEVEL` (no modifier) and `TWO_LEVEL` (only Shift) types. We'll need to add new types in order to allow Hyper and Super as modifiers. Note: the `ALPHABETIC` and `SEMIALPHABETIC` variants of types depends on the way Caps Lock should interact with the key.

Next, there is a `xkb_compatibility` section. We can quite safely ignore it. It may be useful for some fine tuning but this is beyond the scope of this document.

Then comes the very important `xkb_symbols` section. It lists, for each keylabel, what is the type of the key and what are the keysyms it should produce when pressed (in combination with different modifiers). This is where we'll tell XKB what keysym need to be produced with Hyper or Super.

Finally, there is a `xkb_geometry` section. It is quite useless for us. It describes how to draw the keyboard on the screen.

Each of these sections starts with the parts that have been included by XKB when building them. But at this point, this is basically irrelevant. I actually thing that the name of the sections is irrelevant at this point...

If you want to make a first test, it is easy to swap the keysym generated by some keys (in the `xkb_symbols` section) and load the new file to test the result. Beware, if you decide that the 'a' key should now produce the 'b' keysym and do not assign the 'a' keysym to any other key, you have no way to produce it anymore... including for reverting the change... (use copy/paste, or reload the old configuration).

3.3 Adding new modifiers

This is the dirty part, the not totally automatic part where you'll have to thing about stuff...

First, decide which physical keys you want to use as Hyper and Super and use `xev` to get their keycodes. Then, use the `xkb_keycodes` section to get their keylabel. Finally, go into the `xkb_symbols` section, find the description of the key and replace it with whatever you want.

For example, I've decided that the both the left Window key and the 'Document' key should be used for Hyper. Using `xev`, I find that their keycodes are respectively 133 and 100. Looking that in the `xkb_keycodes` section, I discover that the keylabels are respectively `LWIN` (left Window) and `HENK` (used for writing Japanese).

Now, I find the line defining their keysym and replace them. Namely,

```
key <LWIN> {          [          Super_L ] };
```

is replaced by

```
key <LWIN> {          [          Hyper_L ] };
```

and

```
key <HENK> {          [     Henkan_Mode ] };
```

is replaced by

```
key <HENK> {          [     Hyper_R ] };
```

Note that Left Windows was already assigned to Super, changing it to Hyper might have been a poor choice of myself... I've moved Super elsewhere... Anyway, it's done... Think before making your choices.

Go through the same process for all the keys you want to move around. Use `xev` to check that you now have a Hyper and a Super keys (they should produce the correct keysyms).

Note that your Window manager might make use of Super or Hyper in shortcuts. Moving them around might change the behaviour of the WM... You may also take the opportunity to define a proper Meta key separate from Alt, and possibly to place a Compose (`Multi_key`) wherever you want.

Typematrix keyboards need to be switched in 106 mode in order to be able to reprogram the "Shuffle" and "Documents" key (otherwise, the keyboard does produce several keyscans and it's impossible to do anything about it, I think it's a not so good design choice from Typematrix...) This also reprogram 4 others keys that you may want to change back to their usual behaviour... Namely:

`BKSL` ⇒ `XF86Mail`
`AB11` ⇒ `ccedilla` (on a bépo layout)
`HKTG` ⇒ `XF86HomePage`
`AE13` ⇒ `XF86Calculator`

Here also, I chose to change the keylabel to keysym associations rather than the keycode to keylabel association. Mostly because some of these keylabels are positional (`AB11` and `AE13`) and changing them doesn't sound like a good idea.

3.4 Adding new types

Now that we have Hyper and Super keys, we'll need to tell XKB that they are actually modifiers. The good new is that this is already done, the bad new is that they are (probably) both assigned to Mod4.

At the end of the `xkb_symbols` section (just before the `xkb_geometry` one), you should find lines of
`modifier_map Shift { <LFSH> };`
In my case, both `SUPR` and `HYPR` (Super and Hyper) were assigned to Mod4 and nothing was assigned to Mod3. Thus, I simply replace
`modifier_map Mod4 { <SUPR> };`
with
`modifier_map Mod3 { <SUPR> };`
Now, Super and Hyper are both declared as different modifiers (Mod3 and Mod4). If they are assigned as the same modifier, XKB cannot separate them...

Because I also remapped the Caps Lock key to something else, I also had to comment out the line
`// modifier_map Lock { <CAPS> };`
You may want to do the same if you wish to use Caps Lock for something more useful...

Next, go at the end of the `xkb_types` section and add the following type declarations (adapted from the `FOUR_LEVEL_*` types).

```
    type "EIGHT_LEVEL_HYPER_SUPER" {
        modifiers= Shift+LevelThree+Hyper+Super;
        map[Shift]= Level2;
        map[LevelThree]= Level3;
        map[Shift+LevelThree]= Level4;
        map[Hyper]= Level5;
        map[Shift+Hyper]= Level6;
        map[Super]= Level7;
        map[Shift+Super]= Level8;
        level_name[Level1]= "Base";
        level_name[Level2]= "Shift";
        level_name[Level3]= "Alt Base";
        level_name[Level4]= "Shift Alt";
        level_name[Level5]= "Hyper";
        level_name[Level6]= "Shift Hyper";
        level_name[Level7]= "Super";
        level_name[Level8]= "Shift Super";
    };
    type "EIGHT_LEVEL_ALPHABETIC_HYPER_SUPER" {
        modifiers= Shift+Lock+LevelThree+Hyper+Super;
        map[Shift]= Level2;
        map[Lock]= Level2;
        map[LevelThree]= Level3;
        map[Shift+LevelThree]= Level4;
        map[Lock+LevelThree]= Level4;
        map[Shift+Lock+LevelThree]= Level3;
        map[Hyper]= Level5;
        map[Shift+Hyper]= Level6;
        map[Lock+Hyper]= Level5;
        preserve[Lock+Hyper]= Lock;
        map[Shift+Lock+Hyper]= Level6;
        preserve[Shift+Lock+Hyper]= Lock;
        map[Super]= Level7;
        map[Shift+Super]= Level8;
        map[Lock+Super]= Level7;
        preserve[Lock+Super]= Lock;
        map[Shift+Lock+Super]= Level8;
```

```
        preserve[Shift+Lock+Super]= Lock;
        level_name[Level1]= "Base";
        level_name[Level2]= "Shift";
        level_name[Level3]= "Alt Base";
        level_name[Level4]= "Shift Alt";
        level_name[Level5]= "Hyper";
        level_name[Level6]= "Shift Hyper";
        level_name[Level7]= "Super";
        level_name[Level8]= "Shift Super";
    };
    type "EIGHT_LEVEL_SEMIALPHABETIC_HYPER_SUPER" {
        modifiers= Shift+Lock+LevelThree+Hyper+Super;
        map[Shift]= Level2;
        map[Lock]= Level2;
        map[LevelThree]= Level3;
        map[Shift+LevelThree]= Level4;
        map[Lock+LevelThree]= Level3;
        preserve[Lock+LevelThree]= Lock;
        map[Shift+Lock+LevelThree]= Level4;
        preserve[Shift+Lock+LevelThree]= Lock;
        map[Hyper]= Level5;
        map[Shift+Hyper]= Level6;
        map[Lock+Hyper]= Level5;
        preserve[Lock+Hyper]= Lock;
        map[Shift+Lock+Hyper]= Level6;
        preserve[Shift+Lock+Hyper]= Lock;
        map[Super]= Level7;
        map[Shift+Super]= Level8;
        map[Lock+Super]= Level7;
        preserve[Lock+Super]= Lock;
        map[Shift+Lock+Super]= Level8;
        preserve[Shift+Lock+Super]= Lock;
        level_name[Level1]= "Base";
        level_name[Level2]= "Shift";
        level_name[Level3]= "Alt Base";
        level_name[Level4]= "Shift Alt";
        level_name[Level5]= "Hyper";
        level_name[Level6]= "Shift Hyper";
        level_name[Level7]= "Super";
        level_name[Level8]= "Shift Super";
    };
```

The differences are in the way the Caps Lock key is used, depending on whether Levels 1/2 (normal/Shift) and 3/4 (AltGr/Shift+AltGr) are alphabetical characters or not. Here, I choose to treat the four new levels as non-alphabetic, that is they should basically ignore Caps Lock.

3.5 Creating new keysyms

Finally, we can tell XKB to generate the new keysyms. Since it is not possible to actually create keysyms, we need to cheat. We're going to generate Unicode characters (this is possible without restriction). Unicode has some *Private Use Areas* which are points with no characters assigned to them (and the guarantee that it will stay that way). Some of these are used when new glyphs are proposed for possible future inclusion in the standard. Typically, the Tengwar (Tolkien's elf script) characters are currently in a PUA. Anybody is welcome to respect that proposition, but may also ignore it without troubles.

I chose to use the PUA in plan 16, ranging from U100000 to U10FFFD because it seems to be subject of a bit less proposals than the others. And it also allows a nice naming scheme (which would also be

doable with the PUA of plan 15).

Now, each key needs to receive 4 new pseudo keysyms. And as much as possible, I want that to be easy to remember and maintain. Since normal key usually have a "main" symbol which is in ASCII, even ASCII 7 bits (letter or number), I chose to use pseudo keysyms of the shape `U10aall` where `aa` is the ASCII code of the main symbol and `ll` is the level (hence from `05` to `08`) we're defining.

This choice allows to add up to 256 levels (the maximum possible) to each key with the same naming scheme. The main drawback is that it leaves holes in the Unicode range used (typically because ASCII points for uppercase are between number and lowercase, or because many punctuation symbols are not the main symbol of a key). It is, however, easy to remember.

Note also that while it is quite good for letters and numbers, it is not so good with punctuation marks which are not always the main symbol (depending on the layout...)

Anyway, for each of the "normal" keys on your keyboard, you now need to edit its `key` declaration and add the new pseudo keycodes. For example, I now have:

```
key <AD01> {
    type= "EIGHT_LEVEL_SEMIALPHABETIC_HYPER_SUPER",
    symbols[Group1]= [ b, B, bar, brokenbar, U106205, U106206, U106207, U106208 ]
};
...
key <AD04> {
   type= "EIGHT_LEVEL_ALPHABETIC_HYPER_SUPER",
   symbols[Group1]= [ o, O, oe, OE, U106F05, U106F06, U106F07, U106F08 ]
};
...
key <AD06> {
   type= "EIGHT_LEVEL_HYPER_SUPER",
   symbols[Group1]= [ dead_circumflex, exclam, exclamdown, NoSymbol,
                      U102105, U102106, U102107, U102108 ]
};
```

Make sure to preserve the variant (`ALPHABETIC` or `SEMIALPHABETIC`) that was here before (most of these key probably had a `FOUR_LEVEL_*` type).

It may happen that some of these keys only had a `TWO_LEVEL_*` type beforehand (meaning that they ignore AltGr). You can either decide to add new stuff for levels 3 and 4, or simply put a `NoSymbol` keysym.

Note that for the already existing dead keys (notably, `dead_greek` for use in math formulas...), it is also very easy to take advantage of editing this file to just add the keysym somewhere. This one already exists, so no further work should be needed.

Now, the configuration of XKB is complete! After reloading it, Hyper+key and Super+key should work. Since there are no glyphs on these Unicode points, they should probably be displayed as a small rectangle with the corresponding number in it. Use `xev` if you want to double check everything.

4 XCompose

XCompose reads your `~/.XCompose` file (if any), or the system wide one. These files contains lines that tell which sequence of keysym should be turned into which symbol. As said, the `dead_greek` keysym is already handled that way, that is `/usr/share/X11/locale/en_US.UTF-8/Compose` contains lines like

```
<dead_greek> <Y>            : "Ψ"   U03A8    # GREEK CAPITAL LETTER PSI
```

We're going to do the same. Unfortunately, we do not have explicit keysyms names but still want to use them. So we'll have some translation to make.

As a proof of concept, get from `https://http://www.lipn.fr/~moyen/Unicode_keyboard/` the 3 `.pxc` (pre-XCompose) files and place them in a `~/.XCompose-dir/` directory (may be a symlink). If you place them elsewhere, you need to edit `main.pxc` and change the `include` lines to point to the correct files (`%H` stands for `$HOME` and the path must be absolute and not relative to the current directory).

You can look at `fraktur.pxc` and `blackboard_bold.pxc`. They are written in the style of XCompose that we want to manipulate: easy to understand and totally hardware independent (hence shareable). However, the virtual keysyms `dead_fraktur`, `dead_fraktur_bold` and `dead_blackboard_bold` do not exist, so we need to translate these pre-XCompose files into real XCompose files (using the `U10xxxx` pseudo keysyms).

This translation is done by the `translate_xcompose.pl` script (same URL). This is mostly a glorified `sed` which calls itself recursively on `include *.pxc` lines.

First, you may want to change the virtual to pseudo keysyms at the top of the `.pxc` files. Simply replace the `U10xxxx` character by whichever suits you better. These lines must have the shape:
`#PXC <virtual keysym> => <pseudo keysym>`
Especially, they must start with `#PXC`. Extra white space and text is ignored but both the virtual and peusdo keysym must be enclosed in `<...>` and separated by `=>`.

Next, run the script: `translate_xcompose.pl main.pxc`. This should generate three `.xcompose` files (one for each input file); and these files should contain the pseudo keysyms (Unicode points). You only need to run the script on the main file, it will recursively translate any included `.pxc` file.

Finally, edit your own `~/.XCompose` file. It should contain (`%L` is the system wide file, depending on your `locale`):

```
include "%L"

include "%H/.XCompose-dir/main.xcompose"
```

Plus any other personal definitions (or `include`) you may want to use.

Launch a new application (do not clone an existing one), or reboot (rebooting the X server should be enough) to load your new XCompose file, and enjoy.

5 What next

If you want to have access to more special characters, you now just need to create a new `.pxc` file (using explicit virtual keysyms names such as `dead_arrow`, `dead_set_notation`, ...). Don't forget to add a translation directive on top of the `.pxc`. Include the `.pxc` from `main.pxc`, and run the Perl script (always on `main.pxc` as you need to rebuild the inclusion of all subfiles in `main.xcompose`).

The `.pxc` are easily shareable (hence crowdsourceable). Running the script must be done after adding or changing these, but is a minimum amount of work.

6 Legal-ish stuff

This document and the files available at `https://http://www.lipn.fr/~moyen/Unicode_keyboard/` are CC-BY-NC-SA Jean-Yves Moyen, July 2017.

Comes with no warranty. Follow these instructions at your own risk.