

GUADEC 2005

# Towards a localised desktop

*Danilo Šegan*

# 1 • What it takes to get a complete, localised, free software desktop?

Free software has long been considered much better suited for localisation, since anyone is *free* to undertake the job of adjusting it to their own locale preferences.

Any division of entire i10n process is going to be artificial, but we're still going to do it. We're going to make our system first recognize our locale, then enable it to display localised content properly, and finally we need it to let us input such content. With those basics done, we'll turn to describing translation framework from both developers' and users' point of view.

To at least have some direction here, we're going to base our free software desktop on GNU/Linux, X.Org or XFree86 and Gnome. To a varying extent, anything we say will be applicable to other free software systems.

We should also note that any system paths we refer to are based on common installation paths, and hope that readers will be able to correctly deduce their own paths if their packages are installed with differing prefix, datadir, sysconfdir, etc.

# 2 • Related standards

There're many international and local standards which govern the usage and technicalities of i10n support. Lets list only a few of the globally useful ones, which we'll refer to in the text.

ISO 639 lists two-letter ("alpha-2", ISO 639-1) and three-letter ("alpha-3", ISO 639-2) codes for all the worlds languages. Three-letter codes are easier to register and requirements are fairly small: if there's at least 50 documents in a language, it's entitled to a three letter code. Two-letter codes are reserved for "established" languages (see [1] for what is required: lots of documents using this language and official support by an institution). One can apply for ISO 639-1 or 639-2 code at the web pages of USA's Library of Congress [2].

ISO 3166 standard lists two-letter, three-letter and three-digit country codes. One can consult ISO Maintenance Agency web pages [3] directly to find out about relevant code. New codes are assigned only through United Nations procedures.

ISO 10646 or Unicode[4] are synchronized standards which define the "Universal Character Set": a set of all useful characters for the purposes of encoding texts written using any language. They also go as far to define several transformation formats which are directly usable on computers, with UTF-8 (also described in RFC3629) being most interesting to free software desktop users.

ISO/IEC 9945 (available online at [5], commonly dubbed "POSIX", and subset of "Single Unix Specification") is a standard describing portable operating systems and their interfaces, which include interfaces relevant to internationalisation and localisation. Along with ISO 14652 ("Cultural Conventions Technologies", but we'll commonly use the term "locale") and ISO 14651 (defining commonly usable collation table), both available in PDF formats on the web, this standard presents the core of localisation support on free software systems.

Now, anyone can be put off by the sheer amount of standards one would need to consider just to get her locale supported. But fear not, we're going to take shortcuts so we can avoid reading thousands of dull specification pages.

## 3 • Letting the system know about our locale.

libc locales 3

Working with locales 4

First step is to make system acknowledge that what you need is another locale. If it already knows about our locale, then we're already set (and most likely, it already does). But for those unfortunate ones for whom it doesn't know about, or where it has wrong information for them, lets see how to introduce our locale to the system.

### libc locales.

Standard C library (defined by ISO C and POSIX standards), or for short "libc" is what provides many features for handling of special, localised data. It is done through the concept of "locales": collections of data representing a language as used in certain region (commonly a country), with some optional features (eg. using euro as a currency value instead of whatever is default).

Locales provide such information as the metric system used (metric/international or "imperial"), currency values, date and time formats, collation tables (used for sorting), address, telephone, numeric and paper formats, etc.

#### Listing 1.

---

```

1 comment_char %
2 escape_char /
3 LC_TIME
4 day "<U0044><U0061><U0079><U0020><U0031>" ;/
5 "<U0044><U0061><U0079><U0020><U0032>" ;/
6 "<U0044><U0061><U0079><U0020><U0033>" ;/
7 "<U0044><U0061><U0079><U0020><U0034>" ;/
8 "<U0044><U0061><U0079><U0020><U0035>" ;/
9 "<U0044><U0061><U0079><U0020><U0036>" ;/
10 "<U0044><U0061><U0079><U0020><U0037>"
11 END LC_TIME
12 LC_COLLATE
13 copy "iso14651_t1"
14 END LC_COLLATE
15 % Repeat for CTYPE, MESSAGES, MONETARY, NUMERIC, PAPER,
16 % MEASUREMENT, NAME, TELEPHONE, ADDRESS
17 LC_IDENTIFICATION
18 copy "de_DE"
19 END LC_IDENTIFICATION
20 . . .

```

---

Here, we base our locale on de\_DE locale (using copy "de\_DE" directives), and only replace LC\_TIME and LC\_COLLATE definitions. Since our LC\_TIME is very much incomplete, we're going to get many warnings from localedef, but we can ignore them by using -c to force compilation. As for collation, we're using a global table from ISO 14651 standard. ISO 14652 describes some mechanisms on extending it by providing "deltas", but we're not going to go into details.

We can save this in `/usr/share/i18n/locales/testlocale`, and then test using following commands:

**Listing 2.**

---

```

1 # localedef -c -f UTF-8 -i testlocale testlocale.UTF-8
2 # LANG=testlocale.UTF-8 date +%A
3 Day 4

```

---

Petter Reinholdtsen keeps a web page [6] related with GNU libc locales, where you can learn more about writing your own locales. Easiest is to start with other locales.

## Working with locales.

Apart from working with `localedef` command, it's important to understand the basics behind setting one's locale. POSIX systems make use of several environment variables, among which are `LANG`, `LC_ALL` and `LC_category` for each of the categories in a locale. While you can set each of them separately, one most commonly simply sets either `LANG` or `LC_ALL`, and tests her settings with `locale` command.

On GNU systems, one can also use more sophisticated `LANGUAGE` variable, which can hold several fallback languages if first one is not available. It also takes precedence over any other variables on GNU systems.

To fully integrate our newly created locale into system, we want to make X11 aware of it's existence, and we do that by adjusting `/usr/X11R6/lib/X11/locale/locale.dir` and `locale.alias` files.

Finally, we want our default Gnome display manager (GDM) to list our locale as one of the available settings. That can be done by adding appropriate entry to `/etc/gdm/locale.conf`, but it's better to ask for integrating it directly in GDM via Gnome's Bugzilla: this will have added advantage of allowing translation of locale name.

# 4 • Problems of display

Fonts	4
Drawing your text	5
Right-To-Left	5

Once our systems accepts that our language and locale exist, we want it to be able to display it. For some languages (i.e. those based on Latin script), this is mostly trivial. But, in many cases, you're either going to end up with missing fonts or insufficiently capable renderers.

## Fonts.

In most cases, we should be able to find good-enough free software fonts which cover our desired subset of UCS. It's easiest for simpler scripts, such as Latin, Cyrillic and Greek, among others. For Latin scripts, many prefer to use Bitstream Vera fonts, made available thanks to Bitstream Inc. and Gnome Foundation. There're also quite a few Vera derivatives which aim for better coverage, and some of those are DejaVu (covers entire Latin Extended regions and Cyrillic) and Arev (only Sans faces at the time of this writing, adds Greek and Cyrillic). There are other excellent font packages such as URW-CYR (also repackaged as `gsfonts 8.11`) and `Computer Modern Unicode`.

However, if your language requires more special features, you probably need to look for dedicated fonts for your language and/or script. This happens with many complex scripts, so you're better off looking for a font that exactly matches your script.

If you're unable to find a suitable font, you're basically left with only one option: develop your own font. While it may seem daunting at first, there's an excellent tool called `FontForge` [7] which should make it much easier to do.

## Drawing your text.

Finally, once you've got your font, thanks to FreeType2 library, any X program will be able to render glyphs from the font. But that's hardly enough, and we can observe several problems.

First off, we need our system to recognize that this font is suitable for rendering text in our language. On most new free software systems, this is done using fontconfig library (previous "base X fonts" mechanisms were based on the concept of encodings, and long strange "patterns" describing each font and what encoding does it support). Now, fontconfig tries hard to get as close as possible to application-requested font, but if it can't match exactly, it will resort to giving out best font for desired language. It does it's work using a mapping from sets of UCS codepoints to languages, and if it doesn't work correctly for you, you can adjust this mapping yourself.

FontConfig also allows one to tune rendering provided by FreeType2 library, such as turning auto-hinting on or off for certain faces and sizes. It also allows for setting "aliases" so that a request for one font which doesn't exist on a system yields another suitable font.

Since we're basing our free software desktop on Gnome technology, final step in rendering process is going to be Pango: framework for drawing international texts used in Gtk+. Pango includes "shapers" (modules designed for rendering particular languages and/or scripts) for many languages, which includes Arabic, Hangul, Hebrew, Indic, Thai, Tibetan and Syriac at the time of this writing.

## Right-To-Left.

Right-to-left rendering requires special care in many applications. While Pango will properly render any RTL text which is properly tagged, there're more issues to be resolved.

First, we want our entire UI to be "mirrored", which includes menus, toolbars, scrollbars and other UI elements. This is relegated to the toolkit, and Gtk+ handles it using a translation of the message "default:LTR": you have to translate it to "default:RTL" in order to get desired behaviour for RTL languages (see below on how do we do translation itself).

`<center><br /> Untranslated/English Gedit in RTL rendering.</center>`

Most problems arise only when we want to combine RTL and LTR texts in a single paragraph. However, Gnome and Gtk+ have solid foundation for bidirectional texts, so that is unlikely to be a problem in general case, even though it may become a problem in any particular case, depending on the application. For instance, Evolution HTML mail renderer/editor, GtkHTML 3, has only recently had most of the RTL problems resolved.

For some cases where automatic decision making doesn't work, you can manually add specific direction markers by right-clicking the text field, choosing "Insert Unicode control character" from the menu, and selecting appropriate direction mark. This would allow you, for instance, to start your RTL text with an otherwise LTR word (such as "GNOME").

# 5 • Problems of input

Basics of XKB	6
XKB symbol definitions	6
Compose mechanisms	7
Complex input: input methods	8

For most of the world's languages, inputting appropriate text is a matter of simple typing of keys engraved on one's keyboard. X Window System has long offered basic remapping support through xmodmap. Still,

many advantages, especially for i18n, are brought in by using XKB extension. Further, XFree86 4.3.0 introduced "multi-layout" layouts (each layout can be combined with any of the others).

When plain keyboard remapping isn't sufficient, we'll need to look into compositing and input methods.

## Basics of XKB.

Keyboards communicate with the operating system by sending out "keycodes" which indicate what key has been pressed. Mapping between short key designators and numeric keycodes is done in `/etc/X11/xkb/-keycodes/` files: we won't discuss modifying these, since we're assuming that there already are sufficient mappings. You might need to look into these so you could find out what key you want to assign certain functions to.

X (and XKB) communicates with applications in terms of "keysyms" ("key symbols"). Internally, these are just integers assigned a certain meaning. For your copy of X.Org or XFree86, you can find all keysym definitions in `/usr/X11R6/include/X11/keysymdef.h` (you need to strip leading "XK\_" from names). For those Unicode symbols which are not assigned a keysym, you can use format "U" + hexadecimal Unicode codepoint to refer to that symbol. There are also several special keysyms such as "any" and "NoSymbol" (these are equivalent, though). Most common keysyms have names that directly say what they are (eg. "a", "A", "0", "Cyrillic\_a", etc.).

Finally, each key can have a different type: alphabetic keys are behaving differently from numeric or control keys, and that is reflected in their type. For instance, alphabetic keys are affected by "Shift Lock" (better known as "Caps Lock"), while numeric keys are not. Also, each key can have multiple "levels": trivial example of this is a key with two levels: basic-level and shift-level, the latter gotten with pressing and holding down the shift key. How many levels are there, and how do you get them, is defined in `/etc/X11/xkb/types/`. For instance, we'll use `FOUR_LEVEL` and `FOUR_LEVEL_ALPHABETIC` types for our purposes, which provide four levels per each key, and appropriate Shift Lock behaviour.

## XKB symbol definitions.

Multiple different keyboard layouts are termed "groups" in X terminology. One is limited to having a maximum of 4 groups loaded at any one time. In the past, keymaps themselves defined several groups per key, which led to a lot of duplication. Starting with XFree86 4.3.0, all keymaps were reorganized to be "multi-layout": you can now easily merge more than one keymap to fill the 4 available group slots however you please. For compatibility reasons though, this meant moving all keymaps from `/etc/X11/xkb/symbols/` into new hierarchy at `/etc/X11/xkb/symbols/pc/` (note the subdirectory `pc`).

Each symbol file is specific to single language or country, or is shared between several other symbol files through inclusion mechanisms. One symbol file can contain several "variants", with the first one commonly called "basic" and being used when no variant is specified.

For demonstration, lets define a map which outputs Z when one presses AC01 key (usually "A" key on English keyboards; "AC01" means 1st [01] key in 3rd [C] row from the bottom of the keyboard, though this is not necessarily a rule, but only a guideline). Also, we want to merge this with the basic Latin keyboard, and separately define dead and combining accents on third and fourth levels.

### Listing 3.

---

```

1 default partial alphanumeric_keys
2 xkb_symbols "basic" {
3   name[Group1]= "Test keyboard layout";
4   include "pc/latin(basic)"
5   include "pc/test(atoz)"
6   include "pc/test(level3)"
7 };
8 partial alphanumeric_keys
9 xkb_symbols "atoz" {
10  key.type[Group1] = "FOUR_LEVEL_ALPHABETIC";
11  key <AC01> { [ z,      Z,  any,any ] };
12 };

```

---

**Listing 3.** *(continued)*


---

```

13 partial alphanumeric_keys
14 xkb_symbols "level3" {
15   key <AC01> { [ any,any,   dead_acute,      U301 ] };
16 };

```

---

If we save this as `/etc/X11/xkb/symbols/pc/test`, we can test it using `setxkbmap test`.

Lets take a look at what we've got here. First, we define a default variant (to be loaded if no variant is specified). We name it "*Test keyboard layout*", and simply include the `basic` variant from `pc/latin` into it, followed by inclusion of our two subvariants "atoz" and "level3". In subvariant "atoz", we ensure our key type is `FOUR_LEVEL_ALPHABETIC` (meaning, Shift Lock will affect it), and put z and Z on key `AC01`. On third and fourth level we put any, which means that we don't modify the definition there. Finally, in "level3" subvariant, we do the opposite: leave levels 1 and 2 alone, and only define a `dead_acute` and combining acute (Unicode `0x301`) on levels 3 and 4.

We also want our new map to show up in appropriate layout selection tools, such as the one integrated in Gnome. To do that, we need to list it in `/etc/X11/xkb/rules/xorg.xml` or `/etc/X11/xkb/rules/xfree86.xml` (depending on whether we're using X.Org or XFree86). For some applications you might need to update respective `.lst` files instead.

There's also a project started with the wish to become a central place for all keyboard layouts shared between X.Org and XFree86: `xkeyboard-config` [8]. To provide translations for names and descriptions of the all available layouts, one should use Translation Project [9].

## Compose mechanisms.

When we need accented characters, it's easiest to get them if they already exist in UCS. We call such accented characters "precomposed", since the process of attaching an accent to a character is known as "composing". However, in many cases our keyboards are not big enough for all the accented characters we might need, or we don't want to waste precious space on rarely used characters. So, we introduce so called "dead" keys, which modify the following letter we type.

We can find all the available "dead" keys in `keysymdef.h`. Next, we need to define a mapping from dead key and regular character to precomposed character or sequence of characters. This can be done in `/usr/X11R6/lib/X11/locale/en_US.UTF-8/Compose`, or we can define our own X11 Compose file (which we'd need to add to `compose.dir` as well).

Lines in `Compose` files are consisted of whitespace-separated keysyms in angle brackets, followed by colon and quoted UTF-8 string to use as a replacement. Common initial characters are either `Multi_key` or any of the `dead_*` keys.

**Listing 4.**


---

```

1 <dead_acute> <i>      : "i with acute"
2 <Multi_key> <acute> <i> : "i with acute"
3 <dead_acute> <I>     : "I with acute"
4 <Multi_key> <acute> <I> : "I with acute"

```

---

With the above additions or changes to our `Compose` file, whenever we press any of the combinations on the left side, we'll get text on the right side of the colon. We would commonly use a precomposed character there, but if there is no precomposed character we need, we can put a base character and combining diacritic as a string.

However, none of this will work right away in Gtk+ programs. Gtk+ keeps a table of compose combinations compiled in, so you would need to recompile it if you wish to add anything, with the important caveat that Gtk+ only supports many-to-one mappings, and not many-to-many (i.e. right side must be a single character). However, if you choose "XIM - X Input Method" as input method when right clicking on any Gtk+ text field, you will be able to use compose sequences provided by X. You can also set this using environment variable `GTK_IM_MODULE` (value `xim` is for selecting X Input Method).

Another alternative is to directly include combining characters into the symbols file (as we also did in our example above). The difference is that we won't get any precomposed forms without further processing, and we need to input accents after the character, instead of before.

## Complex input: input methods.

For most languages, above mechanisms would suffice. Yet, there are languages which are not suitable for regular input via keyboard. Typical examples are ideographic languages, such as Chinese, Japanese and Korean.

There are several input method libraries available, and most of them can be used with Gtk+ software. Gtk+ also provides internal input methods which are table based, and there are several very simple examples in `modules/input` subdirectory of the Gtk+ source code.

Of popular input method libraries, one should mention XCIN [10] for Chinese, and universal SCIM framework [11].

# 6 • Problem of translation

Translation catalogs: PO files 8

UI translation 9

Documentation translation 9

If we have gotten this far, it means that our free desktop is finally able to display and accept input in the desired language. Still, we're left with mostly English text all around our system.

Most common localisation library in free software is GNU `gettext`. Besides being a developers' library (and being integrated in GNU `libc` as such), it is also a set of tools for working with translation data: `xgettext` for extracting strings from source code, `msgfmt` for compiling translations, `msgmerge`, `msgcat`, `msgfilter` and other tools for managing translations, etc.

## Translation catalogs: PO files.

PO files are *de facto* standard translation file format for free software systems. They are very simple, plain-text files, which allow anyone to use either plain text editor on them, or any of the sophisticated PO editing tools (such as `Gtranslator`, `KBabel`, `POEdit`, `Emacs po-mode`, etc).

All of the GNU `gettext` provided tools either produce or work with PO files.

PO files are consisted of blank-line separated entries which can take two different forms of "messages" for translation: regular and pluralized messages. The difference is that regular forms have only single `msgid` and `msgstr` values (original string and translation thereof), while pluralized forms have two original strings (`msgid` and `msgid_plural` for plurality), and any number of translated strings (`msgstr[i]`, with `i` starting with 0) determined by the "plural-forms" header field.

Initial message in a PO file, a "translation" for empty string "" is called PO file header. It is consisted of metadata such as message encoding, last translator and revision date, plural forms description, language, etc.

Another important aspect for translation is compendia: sets of commonly repeated messages, in easily reusable form. PO files can serve as compendia as well, by using `-C` option to `msgmerge`. This way, translators avoid having to re-translate all the same messages over and over again, by only keeping them separated in a single shared PO file.



## UI translation.

User interface of Gnome Desktop programs comes from several sources: source code (compiled or interpreted), miscellaneous data files (such as .desktop and MIME databases), Glade UI files (though these can be considered source code as well), etc.

`xgettext` from GNU `gettext` tools is able to extract strings only from a subset of necessary files (mostly from source code). For other files, we would need to manage our translations manually. Or perhaps not, since there's `intltool` which unifies translation experience by providing extraction and merging facilities for all the desirable file formats used in Gnome and elsewhere.

`intltool` supports many file formats, among which GConf Schemas, .desktop files, preprocessed XML files, Scheme source code and Glade UI files.

It is very simple to integrate by developers, and even easier to use by translators, who end up caring only about `intltool-update` command, which is their entire interface to UI translation of any software program.

One big problem with UI translation of Gnome-based desktops is that many messages are passed directly from lower-level components over to higher-level components. This usually means that they end up being untranslated. Though it's possible to end up with `libc` messages in the UI, that's not as likely as with ending up with Mozilla/Gecko messages appearing in Epiphany (Gnome web browser). This means that for full translation coverage, you need to work on translation of many of the other, non-Gnome components.

However, there are several issues still remaining. GNU `gettext` merges all exactly the same original strings into one message, which might be a problem since translations might differ depending on the context. This needs to be resolved by developers by providing some context for the message (`glib` provides `Q_` macro for such purposes).

## Documentation translation.

Gnome documentation is written in DocBook XML format. Translating XML files manually is not very hard initially, but it becomes a nightmare if you try to keep up with changes in original documents.

Solution to that problem is to extract only portions of XML documents that are relevant for translation, and ignore the structure and layout. There are several ways to do this, one is to use proprietary software which handles this, other is to try to use `poxml` from KDE project, but recommended way is to use `gnome-doc-utils` and `xml2po`. They are designed especially with Gnome in mind, so they exhibit best performance for Gnome documentation.

`xml2po` provides tools to extract messages for translation into PO files, and later merging of those translations into XML files. `gnome-doc-utils` integrates `xml2po` with the build system, and provides localisation stylesheets for usage in Yelp (the Gnome help viewer).

# 7 • Other problems

Resolving problems we described so far will get us very close to completely internationalised and localised desktop: a goal we set for ourselves. However, there's always more to be done. One important thing is introducing spell checking if it makes sense. Other problem is integration of otherwise incompatible software, which are based on entirely different frameworks. Accessibility for international market is another big issue which is not resolved easily.

## Spell-checking.

Spell checking on free software systems is best done with GNU Aspell [12]. It provides extensive support and documentation for developing your own dictionaries and checking rules.

Unfortunately, some widespread tools (eg. OpenOffice) don't use Aspell, but rather provide their own engines and databases. This means that lot of stuff gets duplicated on our systems. For integration with Gtk+ programs, one should suggest GtkSpell [13].

# 8 • Conclusion

We have gone through most of the steps in getting our free desktop based on GNU, Linux, XFree86 or X.Org and Gnome localised. From time to time, we're going to be confronted with additional problems, but most of them will be simpler to solve once our foundation is right.

And without knowing all of world's languages and cultural needs, we can only hope our foundation is right. If it's not, everyone is invited to contribute, and that is the power free software gives to local groups.

## Appendix: working in restricted environments

Locales and translations	10
Fonts & rendering	11
Keyboard input	11

It's not uncommon to end up in a very restricted environment: you are just a regular user, so you can't touch anything outside your \$HOME. This does not mean that we should be constrained only to locales provided by the system.

### Locales and translations.

Of greatest interest here are LOCPATH and I18NPATH environment variables. LOCPATH points at the directory which contains locales generated with `localedef` (with last argument being directory name inside \$LOCPATH, instead of locale name), and I18NPATH points at input data to be used for generation of locales.

#### **Listing 5.**

---

```

1 ~/locales $ export LOCPATH=~/.locales/
2 ~/locales $ export I18NPATH=~/.locales/
3 ~/locales $ localedef -f UTF-8 -i test ./test@locale
4 ~/locales $ export LC_ALL=test@locale

```

---

Translations path can be set using environment variable NLSPATH, but this won't affect gettext-style translations (it's used only for older "catgets" approach). Unfortunately, there doesn't seem to be a way to do this for GNU gettext using applications. The only alternative is to recompile software on your own, and to set locale path by passing a parameter to `configure` script.

## Fonts & rendering.

Xft2 and FontConfig are commonly set up in such a way to allow users to put fonts for themselves in `$HOME/.fonts/`. It is as simple as dragging and dropping your favourite fonts there.

You can also set `PANGO_RC_FILE` environment variable to a Pango RC file in your home directory, and point `ModuleFiles` to a directory inside your `$HOME`. This should even allow you to use personal Pango shapers.

## Keyboard input.

`setxkbmap` is basically only a front-end to `xkbcomp` which does the actual work of merging a keyboard map into X server. To see what is `setxkbmap` passing over to `xkbcomp`, we can add the `-print` argument:

### Listing 6.

---

```

1 $ setxkbmap test -print
2 xkb_keymap {
3     xkb_keycodes { include "xfree86+aliases(qwerty)"    };
4     xkb_types     { include "complete"                  };
5     xkb_compat    { include "complete+leds(num)+leds(caps)" };
6     xkb_symbols   { include "pc/pc(pc105)+pc/test"      };
7     xkb_geometry  { include "pc(pc104)"                 };
8 };

```

---

This is very useful when we're testing our symbol files, since `setxkbmap` isn't very verbose on error reporting. Thus, we can instead ask it to print keymap definition, and pass that to `xkbcomp -v`:

### Listing 7.

---

```

1 $ setxkbmap test -print | xkbcomp -v -

```

---

At the same time, instead of merging our keymap with X server, we can store it in a compiled (`.xkm`) or source (`.xkb`) file format. For that, we need to use options `-xkm` or `-xkb` to `xkbcomp`.

If we're unable to modify files in `/etc/X11/xkb`, we can either keep `xkb` and `xkm` files in our home directory, and later load them using `xkbcomp`, or we can create a definition based on `setxkbmap -print` output, use relative paths where appropriate, and add our own directory as `xkbcomp -I` argument.

Finally, now that we're able to load XKB, we also want to be able to have our own, personal Compose file. It's very easy starting with XFree86 4.4.0 and recent X.Org versions: by default, file named `/.Xcompose` is loaded as user's compose file (alternately, one can point `XCOMPOSEFILE` variable to any other location). However, this wouldn't be useful enough if one wouldn't be able to base her compose file on already existing compose files. For instance, I'd most likely have something like the following:

### Listing 8.

---

```

1 $ cat ~/.Xcompose
2 include "%L"
3 <dead_grave> <Cyrillic_a>      : "a'"
4 <combining_grave> <Cyrillic_a> : "a'"
5 ...

```

---

Instead of `include "%L"`, which includes default compose file for the active locale, one can put full path to any of the Compose files. Note that the same caveats hold for using this with Gtk+ programs as for the regular Compose files.

## References

- [1] <http://www.loc.gov/standards/iso639-2/criteria1.html>
- [2] <http://www.loc.gov/standards/iso639-2/iso639-2form.html>
- [3] <http://www.iso.org/iso/en/prods-services/iso3166ma/index.html>
- [4] <http://www.unicode.org>
- [5] <http://www.unix.org/version3/>
- [6] <http://www.student.uit.no/~pere/linux/glibc/>
- [7] <http://fontforge.sf.net/>
- [8] <http://freedesktop.org/Software/XKeyboardConfig>
- [9] <http://translation.sf.net/>
- [10] <http://xcin.linux.org.tw/>
- [11] <http://www.scim-im.org/>
- [12] <http://aspell.net>
- [13] <http://gtkspell.sf.net/>