Localizing GNU/Linux and XFree86
A Thailand's Experience

Theppitak Karoonboonyanan
thep@linux.thai.net

January 2004

Abstract

This paper summarizes information gathered by the author during localizing GNU/Linux and XFree86 for Thai. It is by no means a complete reference nor a universal guide for all languages. The author just hopes it is useful for other localizers.

Internationalization is a practice commonly used for making software adaptable to local cultures without modification. Most of this paper discusses this framework and how to localize the system for new locales.

1 Locale

1.1 What is Locale?

Locale is a term introduced by the concept of *internationalization (I18N)*, in which generic frameworks are made so that software can adjust its behaviors to the requirements of different native languages, cultural conventions and coded character sets, without modification nor re-compilation.

Within such frameworks, *locales* are defined for describing particular cultures. Users can configure their systems to pick up their locales. The programs will load the corresponding predefined *locale definition* for working. Therefore, to make internationalized software support a new language or culture, one can create the *locale definition* for it, fill up the required information, and things just work without touching the software code.

The most basic kind of locale in GNU/Linux is the locale for the C library itself. According to POSIX [1], some C functions are defined to be adaptable to locales, such as date and time format, string collation, and so on. GNU C library has implemented all of POSIX locale specifications, plus extensions introduced in ISO/IEC 14652.

In addition to POSIX locale, a localization task most visible to users is message translation. Most free software nowadays uses the framework provided by *GNU gettext,* in which translated messages are complied into a hash database to be looked up at run time. Again, the message database to load is determined by *locale* chosen by users.

1.2 Locale Naming

A locale is described by its language, country and character set. The naming convention as given in OpenI18N guideline [2] is:

$$\langle lang \rangle \_ \langle territory \rangle \, . \, \langle codeset \rangle [\texttt{@} \langle modifiers \rangle]$$

where:

⟨***lang***⟩ is a two-letter language code defined in ISO 639:1988 [3]. Three-letter code in ISO 639-2 [4] is also allowed in the absence of the two-letter version. The ISO 639-2 Registration Authority at Library of Congress [5] has a complete list of language codes.

⟨***territory***⟩ is a two-letter country code defined in ISO 3166-1:1997 [6]. You can get the list of two-letter country codes on-line from the ISO 3166 Maintenance agency [7].

⟨***codeset***⟩ describes the character set used in the locale.

⟨***modifiers***⟩ adds more information for the locale by setting options (turn on flags or use equal sign to set values). Options are separated by commas. This part is optional and implementation-dependent. Different I18N frameworks provide different options.

For example:

- `fr_CA.ISO-8859-1`
  = French language spoken in Canada using ISO-8859-1 character set
- `th_TH.TIS-620`
  = Thai language in Thailand using TIS-620 encoding

If ⟨*territory*⟩ or ⟨*codeset*⟩ is omitted, default values are usually resolved by means of locale aliasing.

1.3 Character Sets

Character set is part of locale definition. It defines what language alphabets are used and how they are encoded for information interchange.

In GNU C library (glibc), locales are described in terms of Unicode. A new character set is described as a Unicode subset, with each element associated by a byte string to be encoded in the target character set. For example, the UTF-8 encoding is described like this:

```
...
<U0041> /x41         LATIN CAPITAL LETTER A
<U0042> /x42         LATIN CAPITAL LETTER B
<U0043> /x43         LATIN CAPITAL LETTER C
...
<U0E01> /xe0/xb8/x81 THAI CHARACTER KO KAI
<U0E02> /xe0/xb8/x82 THAI CHARACTER KHO KHAI
<U0E03> /xe0/xb8/x83 THAI CHARACTER KHO KHUAT
...
```

The first column is the Unicode value. The second is the encoded byte string. And the rests are comments.

As another example, TIS-620 encoding for Thai is simple 8-bit single-byte. The first half of the code table is the same as ASCII, and the second half begins encoding THAI CHARACTER KO KAI (`U+0E01`) at `0xA1`. Therefore, the charmap just looks like:

```
...
<U0041> /x41    LATIN CAPITAL LETTER A
<U0042> /x42    LATIN CAPITAL LETTER B
<U0043> /x43    LATIN CAPITAL LETTER C
...
<U0E01> /xa1    THAI CHARACTER KO KAI
<U0E02> /xa2    THAI CHARACTER KHO KHAI
<U0E03> /xa3    THAI CHARACTER KHO KHUAT
...
```

1.4 POSIX Cultural Conventions

According to POSIX, data of following categories are to be defined so that a number of C functions can adjust their behaviors as per locale:

category	description
LC_CTYPE	character classification
LC_COLLATE	string collation
LC_TIME	date and time format
LC_NUMERIC	number format
LC_MONETARY	currentcy format
LC_MESSAGES	locale messages

1.4.1 Setting Locale

A C application can set current locale with the `setlocale()` function (declared in ⟨locale.h⟩). The first argument is the desired category to set, or `LC_ALL` to set all categories. The second argument is the locale name to choose, or an empty string (`""`) to rely on system environment setting.

Therefore, a typical internationalized C program may call:

```
#include <locale.h>
...
const char *prev_locale;
prev_locale = setlocale(LC_ALL, "");
```

at program initialization, and the system environments are looked up to determine the appropriate locale as follows:

1. If `LC_ALL` is defined, use it as locale name.
2. Otherwise, if corresponding values of `LC_CTYPE`, `LC_COLLATE`, ..., `LC_MESSAGES` are defined, use them as locale names for corresponding categories.
3. For categories that are still undefined by above checks, and `LANG` is defined, use it as locale name.

4. For categories that still undefined by above checks, fall back to `C` (or `POSIX`) locale.

The `C` or `POSIX` locale is a dummy locale in which all behaviors are C defaults (e.g. ASCII sort for `LC_COLLATE`).

1.4.2 LC_CTYPE

LC_CTYPE defines character classification for functions declared in ⟨ctype.h⟩:

```
iscntrl()       isalnum()       isupper()
isgraph()       isalpha()       tolower()
isprint()       isdigit()       toupper()
isspace()       isxdigit()
ispunct()       islower()
```

Since glibc is Unicode-based, and all character sets are defined as Unicode subsets, it makes no sense to redefine character properties in each locale. Typically, LC_CTYPE category in most locale definitions just refer to the default definition (called "i18n").

1.4.3 LC_COLLATE

C functions that are affected by LC_COLLATE are `strcoll()` and `strxfrm()`.

`strcoll()` compares two strings in the same manner as `strcmp()` does. (Note that `strcmp()` behavior *never* changes under different locales.)

`strxfrm()` translate string into a form that can be compared using `strcmp()` to get the same result as directly compared with `strcoll()`.

LC_COLLATE specification is the most complicated one among all locale categories. There is a separate standard for collating Unicode strings, called *ISO/IEC 14651 International String Ordering.* [10] [11] And glibc default locale definition is based on it. You may consider investigating the *Common Tailorable Template (CTT)* defined there before beginning your own locale definition.

Nevertheless, Thai string ordering, despite its well-defined and much-simplified principle, still leaves one exception that makes it require a separate definition: the leading vowel that is put in front of its consonant needs to be reordered before sorting. And as far as I know, Lao is the only other language that suffers from the same situation.

In any case that brings you to defining your own LC_CTYPE definition, you need to know the data structure. In short, the collation is multi-pass. Character weights are defined in multiple levels (four levels for ISO/IEC 14651). Some characters can be ignored (by using "IGNORE" as weight) at first passes and be brought into consideration in later passes for finer judgement.

1.4.4 LC_TIME

LC_TIME allows you to localize date/time strings formatted by the `strftime()` function. You can translate the days of week and months into your own language, define appropriate date and time formats, and even define locale era.

1.4.5 LC_NUMERIC & LC_MONETARY

Some cultures use different conventions for writing number, namely the decimal point, thousand separator and grouping. That is what LC_NUMERIC is for.

LC_MONETARY defines currency symbols used in the locale (reference: ISO 4217 [8]) and how to write monetary amounts.

A single function `localeconv()` in ⟨locale.h⟩ is defined for retrieving information from both locale categories. Glibc provides an extra function `strfmon()` ⟨monetary.h⟩ for formatting monetary amount as per LC_MONETARY, but this is not standard C function.

1.4.6 LC_MESSAGES

LC_MESSAGES is mostly used for message translation purposes. The only use in POSIX locale is the description of yes/no answer. But no standard function has been defined for it yet.

1.5 ISO/IEC 14652

ISO/IEC 14652 Specification method for cultural conventions [9] [11] is basically an extended POSIX locale specification. In addition to the more details in each of the six categories, it introduces six more:

category	**description**
LC_PAPER	paper size
LC_NAME	personal name format
LC_ADDRESS	address codes and format
LC_TELEPHONE	telephone number
LC_MEASUREMENT	measurement units
LC_VERSIONS	locale version

All above categories have already been supported by glibc. C applications can retrieve all locale information using the `nl_langinfo()` function.

1.6 Building Locale

To build a locale, you need to prepare a *locale definition* file describing data for ISO/IEC 14652 locale categories. (Please see the standard document for the file format.) In addition, if you are also defining a new character set, you create a *charmap* file for it, naming every character a symbolic name and describing encoded byte strings as discussed in §1.3.

In general, glibc uses UCS symbolic names (`<U<i>XXXX</i>>`) in locale definition, for convenience in generating locale data for any charmap.

The actual locale data to be used by C programs is in binary form. You must compile the locale definition with the `localedef` command, which accepts arguments like this:

`localedef` [`-f` ⟨*charmap*⟩] [`-i` ⟨*input*⟩] ⟨*name*⟩

For example, to build `th_TH` locale from `th_TH` locale definition file, using `TIS-620` charmap:

```
# localedef -f TIS-620 -i th_TH th_TH
```

You can use the `locale` command with "`-a`" option to check for installed locales and "`-m`" option to list supported charmaps. Issuing the command without argument would show the effects of the environment setting upon locale categories.

2 Overview of GNU/Linux Desktop I18N

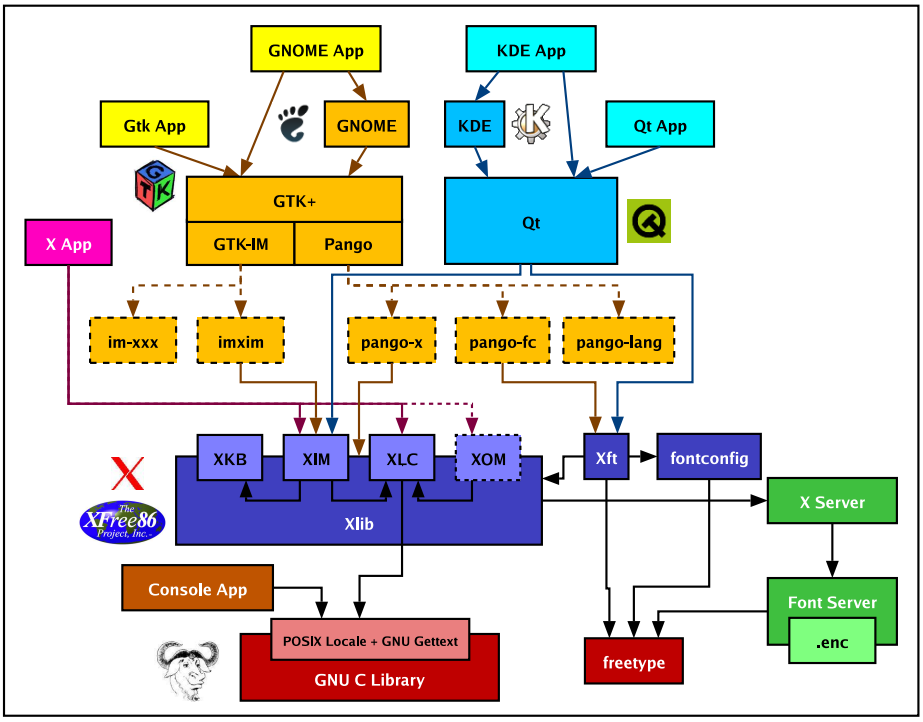

Figure 1: Overview of I18N Tasks for GNU/Linux Desktop

Now that we have discussed the I18N of GNU C library, let's survey further to what make up the whole GNU/Linux desktops in general.

The GNU/Linux desktop is composed of layers of subsystems working on top of one another. In programmer's point of view, the layers could be classified as:

1. **The "C Library" Layer.** As C is the language for developing the UNIX operating system, the lowest level of UNIX API is the C library, no matter in what language your programs are written.

   The C library for GNU/Linux is the GNU C library (glibc), while other OS like *BSD has their own version inherited from the BSD UNIX.

2. **The "X Window" Layer.** As most UNIX systems, X Window provides a network-transparent client-server graphical environment. The X server

takes care of all the hardware tasks, from driving graphics card to keyboard and mouse events. All X applications are clients to the X server. The connections are encapsulated by the X library (aka Xlib) function calls.

Upon GNU/Linux, the only system available for this layer is XFree86, although there has recently been a fork called Xouvert, which may release the first beta soon.

3. **Toolkits.** Writing a program using the low-level Xlib is very tedious, and can be a source of inconsistent GUI when all applications draw menus and buttons by their own preferences. Thus, some libraries are developed as a middle layer, which can help reduce both problems. In X terminology, these libraries are called *toolkits.* And the GUI components they provide, such as buttons, text entry, etc., are called *widgets.*

   Many historic toolkits have been developed along the time, either by the X Consortium like the X Toolkit and Athena widget set (Xaw), or by vendors like XView from Sun, Motif from Open Group, etc. But for the open source world, the major toolkits widely adopted are *GTK+* (the GIMP Toolkit) and *Qt.*

4. **Desktop Environments.** Toolkits are significant tools for creating consistent look-and-feels among the same set of programs. But to make a complete desktop, applications need to interoperate more closely to form a big picture. The concept of *desktop environment* is thus invented to provide common conventions, resource sharing, and communication among programs.

   The first desktop environment ever created on UNIX platforms is *CDE (Common Desktop Environment)* by Open Group, based on its Motif toolkit. But it is proprietary, anyway. The first open source desktop environment for GNU/Linux is *KDE (K Desktop Environment),* based on TrollTech's Qt. However, by some licensing condition of Qt at that time, some developers don't like it. And a second one is created, named *GNOME (GNU Network Object Modelling Environment),* based on GTK+. Nowadays, although the licensing issue of Qt has been resolved, GNOME continues to grow and gets more supports by vendors and its community. As a result, they become the major two desktops widely used on GNU/Linux.

The following sections and the next article in this series will continue the discussion on the I18N frameworks for the subsystems above the C library.

3 I18N in X11R6

In X Version 11 Release 6 (X11R6), Xlib is internationalized via X locales, which is composed of following components [12]:

1. **XLC** – the locale object – provides information that depends on user's language environment, e.g. font set, character set,
2. **XIM** – the input method – manages text inputting,
3. **XOM** – the output method – manages text drawing.

The XLC data definition is composed of following categories [13]:

1. **XLC_FONTSET** defines the font sets (character and font encoding name) used in the locale. This is also used in the XOM.
2. **XLC_XLOCALE** describes character set used in the locale, encoding, conversion, and other attributes.

4 X Input Method

4.1 Keyboard maps

The first step to providing text input for a particular language is to prepare the keyboard map. X11R6 handles keyboard map using the XKB extension.

When you start an X server on GNU/Linux, a virtual terminal is attached to it in raw mode, so that keyboard events are sent from the kernel without any translation.

The *raw scan code* of the key is then translated into *keycode* according to the keyboard model. For XFree86 on PC, the keycode map is usually "xfree86" as kept under `/etc/X11/xkb/keycodes` directory. The keycodes just represent the key position in symbolic form, for further referencing.

The *keycode* is then translated into *keysym* according to the specified layout, such as qwerty, dvorak, or a layout for specific language, chosen from the dataunder `/etc/X11/xkb/symbols` directory. A keysym does not represent a character yet, anyway. It requires Input Method to translate sequences of key events into character, which would be described later. So, an analogy for the meaning of keysym may be just the label screened on the key.

For XFree86, all the above setup are done via the `setxkbmap` command. (Setting up values in `/etc/X11/XF86Config` means setting parameters for `setxkbmap` at initial X server startup.) There are many ways of describing the configuration, as explained in [14]. The default method for XFree86 4.x is by the "xfree86" rule (XKB rules are kept under `/etc/X11/xkb/rules`), with additional parameters:

- *model* – `pc104, pc105, microsoft, microsoftplus, ...`
- *layout* – `en_US, us, th, ...` (For 4.0+, layouts can be mixed up to 64 groups)
- *variant* – (mostly for Latins) `nodeadkeys`
- *option* – group switching key, swap caps, LED indicator, etc. (See Table 1 and 2 for some examples.)

For example:

```
$ setxkbmap us,th -option grp:alt_shift_toggle,grp_led:scroll
```

sets layout using US symbols as the first group, and Thai symbols as the second group. `Alt`-`Shift` combination is used to toggle between the two groups. ScrollLock LED will be the group indicator, which will be on when the current group is not the first group, that is, on for Thai, off for US.

You can even mix more than two languages:

Table 1: XKB Options for Group Switching

Option	Meaning
grp:switch	RightAlt changes group while pressed
grp:lwin_switch	LeftWin changes group while pressed
grp:rwin_switch	RightWin changes group while pressed
grp:win_switch	LeftWin or RightWin changes groupwhile pressed
grp:toggle	RightAlt toggles group
grp:lalt_toggle	LeftAlt toggles group
grp:caps_toggle	CapsLock toggles group
grp:shift_toggle	Both Shifts toggle group
grp:alts_toggle	Both Alts toggle group
grp:ctrls_toggle	Both Ctrls toggle group
grp:ctrl_shift_toggle	Ctrl-Shift toggles group
grp:ctrl_alt_toggle	Ctrl-Alt toggles group
grp:alt_shift_toggle	Alt-Shift toggles group
grp:menu_toggle	Menu toggles group
grp:lwin_toggle	LeftWin toggles group
grp:rwin_toggle	RightWin toggles group
grp:lshift_toggle	LeftShift toggles group
grp:rshift_toggle	RightShift toggles group
grp:lctrl_toggle	LeftCtrl toggles group
grp:rctrl_toggle	RightCtrl toggles group

Table 2: XKB Options for LED Indicator

Option	Meaning
grp_led:num	NumLock LED on if not first group
grp_led:caps	CapsLock LED on if not first group
grp_led:scroll	ScrollLock LED on if not first group

```
$ setxkbmap us,th,jp -option grp:alt_shift_toggle,grp_led:scroll
```

This loads trilingual layout. Alt-Shift is used to rotate among the three groups, that is, Alt-RightShift chooses next group and Alt-LeftShift chooses previous group. The selection wraps up to the other end when selecting next group from the last, and previous from the first. (As a notice, you can see no difference between the effects of Alt-RightShift and Alt-RightShift when only two groups are used.) ScrollLock LED will be on when Thai or Japanese group is active.

The arguments for `setxkbmap` can be specified in `/etc/X11/XF86Config` for initialization on X server startup by describing the `"InputDevice"` section for keyboard, for example:

```
Section "InputDevice"
  Identifier "Generic Keyboard"
  Driver     "keyboard"
  Option     "CoreKeyboard"
```

```
  Option     "XkbRules"   "xfree86"
  Option     "XkbModel"   "microsoftplus"
  Option     "XkbLayout"  "us,th_tis"
  Option     "XkbOptions" "grp:alt_shift_toggle,lv3:switch,grp_led:scroll"
EndSection
```

Notice the last four option lines. It tells `setxkbmap` to use "`xfree86`" rule, with "`microsoftplus`" model (with internet keys), mixed layout of US and Thai TIS-820.2538, and some more options for group toggle key and LED indicator. The "`lv3:switch`" option is only for layout with level 3 (that is, more than normal and shifted keys), which is the case for "`th_tis`" in XFree86 4.4.0. This option sets `RightCtrl` as level-3 shift.

4.2 Providing a Keyboard Map

If the keyboard map for your language is not available, you need to prepare a new one. In XKB terms, you need to prepare a *symbols map* describing keysyms associated to the available keycodes.

The quickest way to start is to read the available symbols files under the `/etc/X11/xkb/symbols` directory. In particular, the usual files used by default rules of XFree86 4.3.0 are under the `pc/` subdirectory. In these files, only one group is defined per file, unlike those old files in the parent directory, in which groups are pre-combined. This is because XFree86 4.3.0 provides a flexible method for mixing keyboard layout.

Therefore, unless you need to support old versions of XFree86, all you need to do is prepare a single-group symbols file under the `pc/` subdirectory.

Here is some excerpt from the `th_tis` symbols file:

```
partial default alphanumeric_keys
xkb_symbols "basic" {
    name[Group1]= "Thai (TIS-820.2538)";
    // The thai layout defines a second keyboard group and changes
    // the behavior of a few modifier keys.

    // converted to THai keysysms - Pablo Saratxaga <pablo@mandrakesoft.com>
    // modified to TIS-820.2538 - Theppitak Karoonboonyanan <thep@linux.thai.net>
    key <TLDE> { [ 0x1000e4f,       0x1000e5b ] };
    key <AE01> { [ Thai_baht,       Thai_lakkhangyao] };
    key <AE02> { [ slash,           Thai_leknung ] };
    key <AE03> { [ minus,           Thai_leksong ] };
    key <AE04> { [ Thai_phosamphao, Thai_leksam ] };
    ...
};
```

Each element in the `xkb_symbols` data, except the first one, is the association of keysyms to the keycode for unshifted and shifted versions, respectively. Here, some keysyms are predefined in Xlib. You can find the complete list in `<X11/keysymdef.h>`. If the keysyms for your language are not defined there, don't panic. You can use the Unicode keysyms instead for XFree86 4.3.0, as

shown in the <TLDE> key entry. (In fact, this should be a more effective way for adding new keysyms.) Just prefix the Unicode value with "0x100" to describe the keysym for a single character.

For more details of the file format, please see [14].

When you finish, you need to regenerate the symbols.dir file so that your symbols file is listed:

```
# cd /etc/X11/xkb/symbols
# xkbcomp -lhlpR '*' -o ../symbols.dir
```

Then, you can try your new layout as described in §4.1.

As an extra addition, you may add your entry to /etc/X11/xkbcomp/rules/xfree86.lst so that some programs like kxkb for KDE can see your layout.

OK, you are happy with the newly fiddled keyboard map. Now you want to include it in XFree86 source. In that case, the all data for XKB are in the xc/programs/xkbcomp subdirectory.

4.3 X Input Method

For some languages, text input is as straightforward as one-to-one mapping from keysyms to characters, such as English. For European languages, a little complication is added by the accents. But for Chinese, Japanese and Korean, the one-to-one mapping is undoubtably impossible. It requires some kinds of keystrokes interpretation to obtain a character.

X Input Method (XIM) is a locale-based framework designed to address the requirements of the text input process of any language. It is a separate service for handling input events as requested by X clients.

Any text entry in X clients is represented by *X Input Context (XIC).* All the keyboard events will be propagated to the XIM, who determines appropriate action for the events based on current state of the XIC, and passes back the resulted characters.

Internally, a common process of every XIM is to translate keyboard scan code into keycode and then to keysym, by calling XKB, whose process detail was described in §4.1. The further processes to convert keysyms into character differ in different locales.

As an example, Thai XIM had been implemented as part of Xlib like those for European languages since X11R6. It was so because Thai input method adds just a little sequence check to the one-to-one mapping. However, some remaining problems with the new XKB model has been fixed in XFree86 4.0.3 and 4.1.0, as documented in [15] and [16]. Thai XIM converts every non-special key event into character by calling XmbLookupString(), a wrapper to XLookupString(). XLookupString() translates the keycode to keysym by the aid of XKB, and the XmbLookupString() wrapper code just translates the keysym into character encoding as per locale. Then, the character is validated with Thai grammar rules in order to determine whether to commit or drop it.

In general cases, however, XIM is usually implemented using the client-server model. The discussion of XIM implementation in more details is beyond the scope of this paper. Please see §13.5 of the Xlib document [17] and the XIM protocol [18] for more information.

In summary, what happen in most XIM-supported applications are:

1. `setlocale(LC_CTYPE, ...)` to set the locale of the XIM.
2. `XSetLocaleModifiers(...)` to let Xlib choose appropriate XIM via the "`@im=...`" X locale modifier.
3. `XOpenIM(...)` to open the XIM.
4. `XCreateIC()` to create an XIC for each text entry.
5. Associate appropriate XIM callbacks to the XIC with the `XSetICValues()` function. Before setting it, the client may query the XIM whether the callback is supported, by using the `XGetIMValues()`, with `XNQueryICValiesList` argument.
6. In the event loop, call `XFilterEvent()` to pass events to XIM filter. If it returns false, the client should discard the event. If it returns true, the client should process the event. For `KeyPress` event, it should lookup string from the key code using either `XmbLookupString()`, `XwcLookupString()`, or `Xutf8LookupString()`.

For step 2, supplying empty locale modifiers (`""`) means to rely on system environment `XMODIFIERS`, so user can specify their favourite XIM, like this:

```
$ export LANG=th_TH.TIS-620
$ export XMODIFIERS="@im=Strict"
```

which specifies `Strict` input method for Thai locale.

5 Font Systems

5.1 Traditional X Font System

Traditional X font system in XFree86 4 supports bitmap fonts (BDF, PCF), Type 1, and, with the aid of FreeType library, TrueType. All are managed at the X server side, either by the X server itself via loadable modules or by a dedicated *X font server (XFS).*

X fonts are referred to by *X Logical Font Description (XLFD),* in which font properties are described, separated by hyphens:

-fndry-fmly-wght-slant-sWidth-adstyl-pxlsz-ptSz-resx-resy-spc-avgWdth-rgstry-encdng

For example:

```
-adobe-times-medium-r-normal--*-100-75-75-p-54-iso8859-1
```

means Times family by Adobe, medium weight, Roman upright shape (no slant), normal width, no additional style, any pixel size, 10.0 points, 75×75 DPI design resolution, proportional spacing, 5.4 pixels average glyph width, using ISO8859-1 character set.

Some fields of the XLFD can be wildcard ('*') to mean the first matched font in the system. For more details of XLFD, please see [19].

To add new fonts to X, you need to prepare a font list file (`fonts.dir`) in the directory using the `mkfontdir` command for bitmap fonts, or `mkfontscale` followed by `mkfontdir` for scalable fonts. After that, you just add the font path to the X server or XFS, depending on your X server setup.

To add font path directly to X server (assuming that appropriate font support modules are loaded at X server startup):

```
$ xset fp+ /your/font/path
$ xst fp rehash
```

To make the fonts preloaded with the X server, add the `FontPath` lines to your `/etc/X11/XF86Config` under the `"Files"` section:

```
Section "Files"
  ...
  FontPath   "/your/font/path"
  ...
EndSection
```

In case you use font server, modify `/etc/X11/fs/config` by adding your font path to the `catalog` line:

```
catalogue = /usr/X11R6/lib/X11/fonts/misc/,...,/your/font/path
```

and then restart xfs:

```
# /etc/init.d/xfs restart
```

5.2 Xft and Fontconfig

XFree86 4 has introduced a number of extensions which brings to a new font system:

- The X Render extension [20] for smooth rendering capability at server side, such as alpha composing, anti-aliasing, sub-pixel positioning and trapezoids.
- The Xft library [21], for X clients to access and rasterize vector fonts at client side by utilizing the FreeType library and the X Render extension.
- The fontconfig library [22], split out of the Xft library for the non-X parts so that all applications including non-GUI programs can share the same font configuration.

As a result, the font management is moved from the server side to client side, which is good for sharing the fonts with other systems outside X, such as the printing process. As a plus, glyph shapes are anti-aliased, resulting in a smoother appearance to the eyes.

Modern desktops, GNOME 2 and KDE 3, have already moved to this new font configuration. The Pango rendering engine has even dropped the traditional X font support in version 1.3 (development version). Therefore, it is supposed to be default method, although the old X fonts are still needed for some existing GTK+ 1.2 applications.

The main configuration file for the fontconfig library is `/etc/fonts/fonts.conf`. The file is an XML data describing font (root) directories and other instructions for font matching and customization. The default configuration in many systems points to system fonts at `/usr/share/fonts/` and user fonts at `~/.fonts/`. Some configuration may also cover the X fonts under the `/usr/X11R6/lib/X11/fonts/` directory. You can check the `<dir>` elements in the file.

Given this configuration, installing new fonts becomes simply copying fonts into one of the defined directories, and then regenerating the font cache using the command:

```
# fc-cache -f
```

viola!

To list all available fonts, you use the `fc-list` command.

References

[1] The Open Group. **The Single UNIX Specification, Version 3.** http://www.unix-systems.org/version3/.

[2] OpenI18N.org. **OpenI18N Locale Name Guideline [Version 1.1 – 2003-03-11].** http://www.openi18n.org/docs/text/LocNameGuide-V11.txt.

[3] **ISO 639:1988 Codes for the representation of names of languages.**

[4] **ISO 639-2:1998 Codes for the representation of names of languages.**

[5] Library of Congress. **ISO 639-2 Registration Authority.** http://lcweb.loc.gov/standards/iso639-2/.

[6] **ISO 3166-1:1997 Codes for the representation of names of countries.**

[7] ISO. **ISO 3166 Maintenance agency (ISO 3166/MA) – ISO's focal point for country codes.** http://www.iso.org/iso/en/prods-services/iso3166ma/index.html.

[8] **ISO 4217 Currency and funds codes.**

[9] **ISO/IEC 14652 Specification method for cultural conventions.**

[10] **ISO/IEC 14651 International string ordering.**

[11] **ISO/IEC JTC1/SC22/WG20 - Internationalization.** http://anubis.dkuug.dk/jtc1/sc22/wg20/.

[12] Katsuhisa Yano and Yoshio Horiuchi. **X11R6 Sample Implementation Frame Work.**
`xc/doc/hardcopy/i18n/Framework.PS.gz`.

[13] Yoshio Horiuchi. **X Locale Database Definition.**
`xc/doc/hardcopy/i18n/LocaleDB.PS.gz`.

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

[15] Theppitak Karoonboonyanan. **XFree86 Thai Supports.**
http://linux.thai.net/thep/th-xwindow.

[16] Theppitak Karoonboonyanan. **Thai Input Method Implementations.**
http://linux.thai.net/thep/th-xim.

[17] James Gettys and Robert W. Scheifler. **Xlib – C Language X Interface.** X Consortium Standard, X Version 11, Release 6.4.
`xc/doc/hardcopy/X11/xlib.PS.gz`.

[18] Masahiko Narita and Hideki Hiura. **The Input Method Protocol Version 1.0.** X Consortium Standard, X Version 11, Release 6.4.
`xc/doc/hardcopy/XIM/xim.PS.gz`.

[19] Jim Flowers. Stephen Gildea (Edit). **X Logical Font Description Conventions Version 1.5.** X Consortium Standard, X Version 11, Release 6.4.
`xc/doc/hardcopy/XLFD/xlfd.PS.gz`.

[20] Keith Packard. **Design and Implementation of the X Rendering Extension.** *FREENIX Track, 2001 Usenix Annual Technical Conference,* Goston, MA, June 2001. USENIX.
http://keithp.com/ keithp/talks/usenix2001/.

[21] Keith Packard. **The Xft Font Library: Architecture and Users Guide.** XFree86 Technical Conference, The XFree86 Project, Inc., October 2001. Usenix Associatation.
http://keithp.com/ keithp/talks/xtc2001/.

[22] Keith Packard. **Font Configuration and Customization for Open Source Systems.** GNOME Users And Developer European Conference, 2002.
http://keithp.com/ keithp/talks/guadec2002/.