Localizing GTK+

Theppitak Karoonboonyanan
thep@linux.thai.net

January 2004

Abstract

This paper summarizes information gathered by the author during contributing to Pango and GTK+ projects and developing own Thai modules [1] [2]. It is by no means a complete reference nor a universal guide for all languages. The author just hopes it would be useful for other localizers.

GTK+ 2 is more than internationalized. It is indeed a multilingual system. Language supports are modularized for input methods, text drawing and language processing. This paper describes the framework in localizer's point of view.

1 GTK+ I18N Framework

1.1 I18N or Multilingual?

Like other modern Unicode-based toolkits, GTK+ [3] is more than internationalized. In fact, it is multilingual, that is, it supports multiple languages and scripts at a time, instead of just the locale's language and script like traditional I18N. This is simply the nature of Unicode, where characters of all scripts in the world are encoded in a single code table. To support Unicode means to support all scripts.

1.2 GTK+ I18N Framework

GTK+ multilingual support is acheived by dynamic I18N. That is, it is modularized so that language support modules are dynamically loaded on demand, rather than trying to cover everything in a single place. As a result, localization is flexible, and side-effect bugs between different language supports are minimized.

There are two I18N frameworks in GTK+ 2:

- **GTK+ IM** – dynamically selectable input method framework based on pure GTK+ API
- **Pango** – a quality text layout engine, which analyzes and shapes text portions by loading and calling corresponding modules as per their Unicode ranges. There are two kinds of Pango engines: language engines and shaping engines.

2 GTK+ Input Method Modules

2.1 GTK+ Input Method Modules

As a cross-platform toolkit, instead of just relying on platform input methods, GTK+ 2 defines its own framework using pure GTK+ API's. This provides high-level abstraction, making input methods development a lot easier than writing XIM servers. (GTK+ can still use the several existing XIM servers through the **imxim** bridging module, anyway.) Besides, the input methods developed become immediately available to GTK+ in all platforms it supports, including XFree86, Windows, and GNU/Linux framebuffer console. The drawback is that the input methods cannot be shared with non-GTK+ applications.

The *Internet/Intranet Input Method Framework (IIIMF)* [4] defined and developed by OpenI18N.org allows input methods to be shared among several kinds of applications, or even across platforms connected in the same network. The details of IIIMF is beyond the scope of this paper, anyway.

2.1.1 Client-Side Coding

A quick example for client-side code for text input using GTK+ IM from which you can study is the **GtkEntry** itself. GTK+ IM clients usually handle text entries using the **GtkIMMulticontext** class, which provides dynamic IM selection via menu items queried from GTK+ IM modules list.

Technically speaking, **GtkIMMulticontext** is derived from the **GtkIMContext** base class, which provides interface for all IM context implementations. When user selects a new input method menu item, the corresponding IM context subtype is created by calling the `im_module_create()` function of the corresponding module. The newly created IM context becomes the slave of **GtkIMMulticontext** and takes care of all the interfaces for it.

In client's point of view other than the construction process described above, all IM contexts are accessible via the **GtkIMContext** interface. An important interface for text input is the `gtk_im_context_filter_keypress()` function. The client would call it upon key press to pass the event to the input method. If it returns TRUE, that means the input method has consumed the event, and the client should discard it.

There are also interfaces for the other direction. Input method can call the client for some action by emitting GLib signals, for which the handlers may be provided by the client by connecting callbacks to the signals:

"preedit_changed"

> Uncommitted (preedit) string is changed. Client may update the display, but not the input buffer, to let user see the keystrokes. Preedit string can be retrieve using the `gtk_im_context_get_preedit_string()` function.

"commit"

> Some characters are committed from IM. The committed UTF-8 string is also passed as argument, from which the client can take into its input buffer.

"retrieve_surrounding"

The IM wants to retrieve some text around the cursor. Client should return the context string as much as possible to the IM by using the `gtk_im_context_set_surrounding()` function.

"delete_surrounding"

The IM wants to delete text around cursor. Client should delete text portion around the cursor as requested.

We shall return to these signals later when we talk about input method implementation.

2.1.2 IM Implementation

GTK+ input methods are implemented in loadable modules providing following entry functions:

`void im_module_init(GTypeModule *module)`

initialization – normally registers the IM context type (as a **GtkIMContext** derivative)

`void im_module_exit()`

module clean-ups upon unloading

`void im_module_list(const GtkIMContextInfo **ctxs, int *n)`

lists information of all IM's provided in the module

`GtkIMContext *im_module_create(const gchar *context_id)`

creates **GtkIMContext** instance

The main task of the IM module is to define new IM context class or classes by extending the **GtkIMContext** base class and override some virtual functions. (Please see GObject Reference Manual [5] for new type derivation.) Usually, a virtual function always overridden is `filter_keypress()`, which has following prototype:

```
gboolean (*filter_keypress) (GtkIMContext *context,
                             GdkEventKey  *event);
```

The function is to be called by the client upon key press event. It can determine proper action to the key and return TRUE if it means to consume the event or FALSE to pass the event back to the client.

Some IM (e.g. CJK and European) may do a stateful conversion by incrementally match the input string with predefined patterns until unique pattern is matched before committing the converted string. During the partial matching, the IM emits the `"preedit_changed"` signal to the client for every change, so that it can update the preedit string to the display. Finally, to commit characters, the IM emits the `"commit"` signal, associated with the converted string as argument, to the IM context. Thus, the client must handle the `"commit"` signal to catch the input text.

Some IM (e.g. Thai) is context-sensitive. It needs to retrieve text around the cursor to determine appropriate action. This can be done through the

`"retrieve_surrounding"` signal. The IM calls `gtk_im_context_get_surrounding()` function to get the context. The function would then emit the signal to the IM context, which should handle it with a function supplied by the client. The handler should read client's text buffer as requested and reply using the `gtk_im_context_set_surrounding()` function call.

In addition, the IM may request to delete some text from the client's input buffer (as required by Thai advanced IM which also corrects the illegal sequences, for example). This can be done via the `"delete_surrounding"` signal, which is emitted from the IM by calling the `gtk_im_context_delete_surrounding()` function. So, the client should also catch the signal to support the case.

3 Pango Engines

3.1 Pango Overview

Pango [Gk *pan* all + Jap *go* language] [7] is a multilingual text layout engine designed for quality text typesetting. Although it is the text drawing engine of GTK+, it can also be used outside GTK+ for other purposes, such as printing [8].

This section will discuss the bird-eye view of Pango as necessary for localizers. You may consult Pango reference manual [9] for deeper details.

3.1.1 PangoLayout

At the high level, Pango provides the **PangoLayout** class which takes care of typesetting text in a column of given width, as well as other information necessary for editing, such as cursor positions. Its features may be summarized as follows:

1. **Paragraph Properties.**
   - indent
   - spacing
   - alignment
   - justification
   - word/char wrapping modes
   - tabs
2. **Text Elements.**
   - get lines and their extents
   - get runs and their extents
   - character search at (x, y) position
   - character logical attributes (is line break, is cursor pos, etc.)
   - cursor movements
3. **Text contents.**
   - plain text
   - markup text

3.1.2 Middle-level Processing

Pango also provides access to some middle-level text processing functions, although most clients in general do not use them directly. To grab a brief understanding of Pango internal, let's discuss some highlights.

There are three major steps for text processing in Pango[1]:

1. `pango_itemize()`

   breaks text into chunks (items) of consistent direction and shaping engine. This usually means chunks of texts of the same language with the same font. Each chunk is represented by a **PangoItem** instance, which associates with it the corresponding shaping and language engines through its **PangoAnalysis** member.

2. `pango_break()`

   determines possible line, word and character breaks within the given chunk of text (i.e. within the given **PangoItem**). It calls the language engine of the chunk (or the Unicode-based `pango_default_break()` if no language engine exists) to fill the **PangoLogAttr** array which describes logical attributes of the characters (is-line-break, is-char-break, etc.).

3. `pango_shape()`

   converts text chunk into glyphs, with proper positioning. It calls the shaping engine of the chunck (or the default shaping which is currently suitable for European languages) to obtain a **PangoGlyphString**, which describes the information of the glyphs to render (code point, width, offsets, etc.).

3.2 Pango Engine Implementation

By the time this paper is written, the latest stable version of Pango is 1.2.5. However, there are a lot of changes from this version in current CVS and development versions. To be useful for new modules creation, our discussion below shall be based on Pango 1.3.x series.

Pango engines are implemented in loadable modules which provide following mandatory entry functions:

`void script_engine_init(GTypeModule *module)`

initializes the module – usually registers the types of the engines provided in the module.

`void script_engine_exit()`

module clean-ups upon unloading.

`void script_engine_list(PangoEngineInfo *engines, int *n_engines)`

lists information of all engines provided in the module. The engine information is described as **PangoEngineInfo** struct, containing the engine name, engine type (language/shaping), render type (for shaping engine only), and the scripts (languages) it supports.

[1] This is a very rough classification. Obviously, there are further steps, e.g. line breaking, alignment, justification. Discussing all of them here is out of localization interests.

```
struct _PangoEngineInfo
{
  gchar *id;
  gchar *engine_type;
  gchar *render_type;
  PangoEngineScriptInfo *scripts;
  gint   n_scripts;
};
```

Please see <pango/pango-engine.h> and <pango/pango-script.h> for details.

`PangoEngine *script_engine_create(const char *id)`

creates a **PangoEngine** instance for the given ID. It should return NULL for unknown ID.

In Pango 1.3.*x*, Pango engines are defined as a new type derived from either **PangoEngineLang** or **PangoEngineShape.** You define a new engine by extending the base type and overriding its virtual functions, as will be discussed below.

3.3 Pango Language Engines

As discussed in §3.1.2, Pango language engine is called to determine possible break positions in a chunk of text of a certain language. All Pango language engines are instances of types derived from the **PangoEngineLang** base class. The virtual function to override is:

```
void (*script_break) (PangoEngineLang *engine,
                      const char      *text,
                      int              len,
                      PangoAnalysis   *analysis,
                      PangoLogAttr    *attrs,
                      int              attrs_len);
```

The task of the `script_break()` function is to fill the given **PangoLogAttr** array with attributes of every character in the text. The logical attributes for a character are following flags:

```
struct _PangoLogAttr
{
  guint is_line_break : 1;
  guint is_mandatory_break : 1;
  guint is_char_break : 1;
  guint is_white : 1;

  guint is_cursor_position : 1;

  guint is_word_start : 1;
```

```
  guint is_word_end   : 1;

  guint is_sentence_boundary : 1;
  guint is_sentence_start : 1;
  guint is_sentence_end : 1;

  guint backspace_deletes_character : 1;
};
```

And the meaning of the flags are as follows:

Flag	Description
is_line_break	can break line in front of the character
is_mandatory_break	*must* break line in front of the character
is_char_break	can break here when doing character wrap
is_white	is white space character
is_cursor_position	cursor can appear in front of character
is_word_start	is first character in a word
is_word_end	is first non-word character after a word
is_sentence_boundary	is inter-sentence space
is_sentence_start	is first character in a sentence
is_sentence_end	is first non-sentence character after a sentence
backspace_deletes_character	backspace deletes one character, not entire cluster *(new in Pango 1.3.x)*

3.4 Pango Shaping Engines

As discussed in §3.1.2, Pango shaping engine serves the conversion of characters in a text chunk of a certain language into glyphs, as well as their positioning according to the script constraints. All Pango shaping engines are instances of types derived from the **PangoEngineShape** base class. The virtual function to override is:

```
void (*script_shape) (PangoEngineShape *engine,
                      PangoFont        *font,
                      const char       *text,
                      int               length,
                      PangoAnalysis    *analysis,
                      PangoGlyphString *glyphs);
```

The task of the `script_shape()` function is to fill the given **PangoGlyphString** buffer with the converted glyphs information. To see what a shaping engine needs to provide, let's examine the structure of **PangoGlyphString** a bit:

```
struct _PangoGlyphString {
  gint num_glyphs;
  PangoGlyphInfo *glyphs;
  gint *log_clusters;
```

```
  /*< private >*/
  gint space;
};
```

In the structure, `glyphs` and `log_clusters` are parallel arrays of `num_glyphs` elements, storing information of the glyphs sequence to render from left to right. (So, for RTL scripts, the glyphs sequence is just reverted.) The `glyphs` array describes the glyphs themselves, with their positionings, while the `log_clusters` maps the glyphs back to characters in the original text.

Each element of the `glyphs` array describes a glyph with **PangoGlyphInfo** type:

```
struct _PangoGlyphInfo
{
  PangoGlyph    glyph;
  PangoGlyphGeometry geometry;
  PangoGlyphVisAttr  attr;
};
```

where `glyph` is the glyph index within the font, `geometry` is the width and positioning of the glyph, and `attr` describes additional visual attributes of the glyph.

Glyph `geometry` of **PangoGlyphGeometry** contains following information:

```
struct _PangoGlyphGeometry
{
  PangoGlyphUnit width;
  PangoGlyphUnit x_offset;
  PangoGlyphUnit y_offset;
};
```

The `x_offset` and `y_offset` here describe the relative shifting of the glyph in reference to previous glyph. So, you can adjust your glyph positioning here.

Finally, there is only one additional visual attribute of the glyph for the time being:

```
struct _PangoGlyphVisAttr
{
  guint is_cluster_start : 1;
};
```

As its name describes, `is_cluster_start` determines if the glyph is a cluster starter (useful for Arabic, maybe).

References

[1] **TLWG LibThai.**
http://libthai.sourceforge.net/.

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

[3] **GTK+ – The GIMP Toolkit.** http://www.gtk.org/.

[4] **IIIMF Project.** http://www.openi18n.org/subgroups/im/IIIMF/.

[5] **GObject Reference Manual.** http://developer.gnome.org/doc/API/2.0/gobject/.

[6] **GTK+ Reference Manual.** http://developer.gnome.org/doc/API/2.0/gtk/.

[7] **Pango.** http://www.pango.org/.

[8] **Pango – Design Goals.** http://www.pango.org/design.shtml.

[9] **Pango Reference Manual.** http://developer.gnome.org/doc/API/2.0/pango/.