I18N : how to

Georges Khaznadar <georgesk@ofset.org>

lycée Jean Bart – Dunkerque/OFSET

October 2007

1. GNU Gettext : a full-featured system
   - I18N
   - L10N
2. A case study
   - The initial package
   - Marking translatable strings
   - Adding Gettext capabilities
   - Making the first localisation
   - Automating further development cycles
3. The developer's point of view
4. The translator's point of view
5. Available tools
6. Other package types

GNU Gettext is the right tool to make efficient translations of free software pieces. It allows to split the work in two parts:

1. The $I\overbrace{nternationalisatio}^{18}N$: **I18N**
2. The $L\overbrace{ocalisatio}^{10}N$: **L10N**

The internationalisation part is controlled by the developers, the localisation is the job of people having good linguistic skills.
Of course, some skills can overlap.

The internationalisation job must be done once for a given software package. It is preparing the sources to be translated, without translating them. The key work is to **mark the strings** which have to be translated. It is the first stage in using GNU Gettext.

1st example
```
sprintf(buf, "%s", varname);
```
no markup needed since the string has just a functional value.

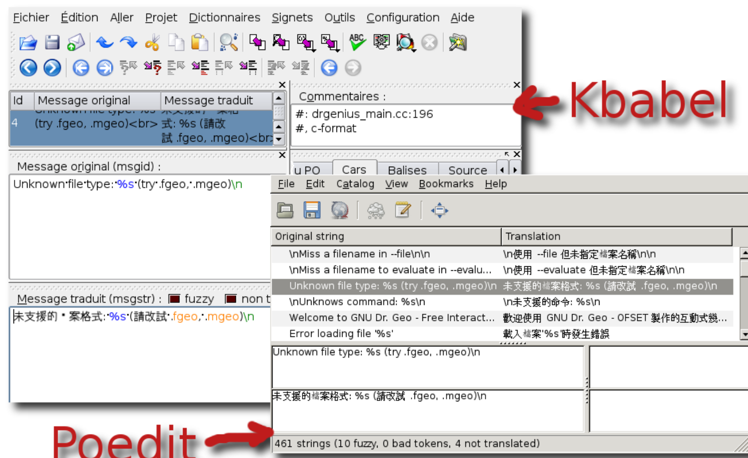
2nd example
```
errmsg("Can't open file.");
```
needs a markup, since this string is intended to be read by users. Here is the markup technique:
```
errmsg(_("Can't open file."));
```

The second stage in I18N is to define the construct used to mark the strings, like in `_("the string to translate")`, i.e. the `_( )`, as something valid in the language used for the software's sources (C, C++, Python, Perl, Ada, etc.)
It is often defined either as a macro or as a function. Some lines are added in the sources to link the program to the GNU Gettext library, and to catch the current *locale* from the environment.

The localisation is a work for people with good linguistic skills, it can be performed in a friendly environment, like Emacs Poedit or Kbabel. Here is a non exhaustive list of features of these environments:

- View the original strings and their translations
- Go to the next untranslated string
- Go to the next fuzzy translation
- See the contexts for this string in the sources
- See other translators's work in the same context (good translators often master more than two languages).

Some screenshots: Poedit, Kbabel.



The package timecalc

Timecalc is an application to compute as accurately as possible timestamps and delays, and taking in account as well as possible fuzzy time units like one month, one year, etc. It is (c) 2000-03 by Jean-Pierre VERRUE, distributed under the GPL license.
Here is a typical string which may be translated to other languages, ine the C source code :

```
if (tm->tm_mday > day_tab[leap][tm->tm_mon])
        {
        if (display_warnings)
                fprintf(stderr, "Day of month doesn't match (%d/%d). Altered (%d
/%d)\n",
```

The initial package is written in C language, with English messages and informational character strings. This is a good start point, since Gettext requires that the identifiers used for the translations be pure ASCII strings.

The author, Jean-Pierre VERRUE, is French. Had he written the messages in French, which uses some non-ASCII characters, the first step would have been to replace them by pure ASCII identifiers.

Running `etags`

The application `etags` allows to create a "TAGS" file usable by Emacs (if you prefer vi as a text editor, use `ctags`).
Just run once the following command: `etags *.c`

Deciding which strings need translation

Here is a recipe:

- With Emacs, edit a new file named `template.po` in the same directory than the TAGS file.
- Use the keyboard shortcut , (comma) to find the next candidate string.
- When the string must be translated, use the keyboard shortcut **Alt+,** (press down the Alt key then hit the comma ... this combination is named M-, in Emacs jargon)

When you press **Alt+,** two thing happen at the same time: the original message is added in the file `template.po` and the original string in the source file is marked.

```
#: /tmp/timecalc-0.11.1/timecalc/main.c:182
msgid "%s line:%d "
msgstr ""
```

Figure: The message is added in `template.po`

```
        fprintf(stderr, _("%s line:%d "), __FILE__, __LINE__) ;
```

Figure: The message is marked in the C source file

The following steps are a summary from GNU's documentation about gettext: see GNU's website, `http://www.gnu.org/software/gettext/`

First : Import the gettext declaration and create a simple macro

```
#include <libintl.h>
#define _(String) gettext (String)
```

Then: Trigger gettext operations in the main function:

```
    setlocale (LC_ALL, "");
    bindtextdomain (PACKAGE, LOCALEDIR);
    textdomain (PACKAGE);
```

Here are five steps to manage a localisation. Please notice that the macros LOCALEDIR and PACKAGE shold expand to useful values, for example respectively /usr/share/locale and timecalc.

- Copy the file template.po to a localisation file, for example zh_TW.po
- use Emacs, Poedit or Kbabel to translate the strings
- compile the localisation file with "msgfmt" to make a binary file
- upon install, copy this binary file under the "LOCALEDIR", with the name "PACKAGE.mo"

To make things happen faster after this i18n work, it is worth creating a separate directory, named po, move every gettext stuff into it, and define some makefile which can:

- grab every new marked string from the sources, thanks to the command xgettext
- merge the new strings which appear in template.po into every maintained localisation file
- compile the localisation files after they are updated and install the binaries in the right place.

When the i18n work is done, the developers just need to mark the new strings with **_()** whenever they should be translated, then release the new versions, and send a message to the language teams to announce the new release.
It is possible for the eveloper to fiddle with on-the-fly language choices, juste by modifying slightly the usage of the function `setlocale` in the main function. Beware: for it to be possible, the language variant choosen on-the-fly must exist in the currently intalled locales.

The translators have to do the first full translation when they receive the first i18nised release of a program. Later, they just need to watch the annunces for new releases and check whether new translatable strings have been added.
It is possible to merge a localisation PO file with a compendium of already translated frequent sentences (like save file, open file, quit, save as, etc.) Then the system wil create "fuzzy" translations, which just need to be reviewed, and eventually fixed by the translator.

Available tools

- **etags** analyse the structure of sources and prepare the markup
- **emacs** features a powerful tool to mark the right strings and gather them into a template for PO files.
- **xgettext** extracts every marked strings and updates the po files
- **msgmerge** merges PO files, enventually creating fuzzy translations
- **msgfmt** compiles PO files to make MO files

Other package types

GNU Gettext is suitable for a variety of languages : C++, Objective-C, sh script, bash script, Python, GNU CLISP, Emacs Lisp, librep, GNU Smalltalk, Java, GNU awk, Pascal, wxWidgets , Tcl, Perl, PHP, Pike, Ruby, and R.
The utilities may vary for each of these languages, so the developers must remain adaptative.
However, the translation teams just need to deal with PO files, independently frm the language of the sources of the package, sot they keep their usages and they favorite tools and can be efficient.