Bilingual Programming

Second language

Linux is international. It was started by a programmer from Finland who speaks Swedish. Aided by a Welsh-speaking lieutenant. Supplemented with a kernel maintainer from Brazil. So why is all our software written in English? This month multi-lingual development and the *gettext* package. BY STEVEN GOODWIN

The English language holds the same power in today's society that Latin did many hundreds of years ago. It's not the most expressive language, nor is it the most popular. It certainly isn't the easiest to learn. It is, however, the most widespread. With the remnants of the old British Empire still present, and the continued growth of America, people are required to use English in order to compete on the world stage.

Computers and the Internet have increased this linguistic strangle-hold. More web pages exist in English than any other language. More programming languages use English words like *if* and *while*, regardless of the designer's nationality. Most software uses prompts and error messages that are written in English.

However, with Linux taking control of many different systems across the globe, it would appear to be xenophobic of us to continue developing 'English-only' software. Adding the ability to change the language (or *locale*) of your software is not a difficult task to achieve, but it shows a wider commitment to your project, and the open source community in general. Even if you can not translate the text yourself, you can make it easier for someone else to do so by following the guidelines in this article.

Turning Japanese

GNU/Linux uses a technique known as *locales* to determine many things: the appropriate translations for text, the character set required to represent the alphabet, and cultural specifics like the expression of numbers, or the date. Each area is considered in the box: Locale Categories, although the focus of this article will be on text translation.

So let us start with the simplest program we know, *Hello World*. We shall be coding in C, although the same techniques can be applied regardless of language. You'll be able to test the PHP equivalent by using the code in box: In PHP.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  printf("Hello World!\n");
  return 0;
}
```

It's fairly obvious to us where the translation string will need to go. At compile time, however, we do not know what the replacement string will be, or what languages it will need to be in. This prevents us from including any translation data directly into program. Instead, we must build up *catalogs* of each word and phrase used by our program, and employ the *gettext* package to act like a dictionary. This will replace our (English) words with the correct foreign version at run time. What is 'correct' will be determined by the user's specific locale.

We are required to do two things,

1. Mark the source code to say 'get me the correct words for the phrase XYZ'
2. Build a translation dictionary for each language we need to support

Marking the source is a simple process. We, as programmers, must work through each line of the code and indi-

In PHP

Writing multi-lingual software in PHP is no different from using C. The functions even have the same name! However, when run as part of a web page, it might be more suitable to specify the locale explicitly. Perhaps coming from an session variable, or cookie on the users machine.

```
<?php
setlocale(LC_ALL, "fr_FR");
textdomain("lm");
echo gettext("Hello World!\n");
?>
```

The effect of *setlocale* can also be achieve by using the *putenv* function.

```
putenv ("LANG=fr");
```

cate which lines of text will need translating. We can do this by calling a special function (called, not surprisingly, *gettext*) that will consult the dictionary and convert our string to something suitably foreign.

```
printf(gettext("Hello World!\n"));
```

This function can be found in the *libintl* header file, so we must,

```
#include <libintl.h>
```

Compiling under GNU/Linux requires no extra link libraries for the code to work. The word GNU is essential here. That is because the internationalization features are included directly in *glibc*. Users of other Unix-like systems may not be so lucky. However, without a language catalog, no translations will be made. That doesn't matter at the moment, since the English text will be output in all cases where a translation can not be found. C programmers will also note that this method is not all-encompassing, because there is more than one way to declare a string. However, we've only learnt one way to mark strings for translation. So we will need to use another method, to cope with those cases where a function call to *gettext* would result in a syntax error. For example,

```
char *pHello = "Hello, World!\n";
```

To circumvent this problem, we need to create a macro that includes a marker, but has no adverse effect on the syntax.

```
#define gettext_noop(String) String
...
char *pHello = gettext_noop("Hello, World!\n");
```

We then need to invoke the translation module in the usual way, before we output the string. Like so,

```
printf (gettext (pHello) );
```

These markers not only perform the translation when the program is running, but indicate to us what text needs to be translated. We shall shortly see a tool that makes use of these markers itself to help build the dictionary of translations. If we were to build the dictionary manually (but why would we?!), the *gettext_noop* marker would be unnecessary.

Some programmers prefer to replace this nine character marker with a single character macro, such as the underscore. This is because the word *gettext* (and both brackets) can cause many lines to break the 80 character limit. This is simply,

```
#define _(str) gettext (str)
#define N_(str) gettext_noop (str)
```

The GNU standard prefers a space between function name and bracket, but this is often omitted.

We can now move on and build our foreign language dictionary.

Vienna Calling

Building a file that contains all the strings in a program is not as time-consuming as you might think. Naturally, it is a very common task, and can be achieved by using a tool named *xgettext*. This is one of the few instances where the 'x' does not stand for an X Window program. Instead, it is short for 'extract'. This program will search the source file for any string used in conjunction with the function call *gettext* (or *gettext_noop*) and place the text into a catalog file (ending the suffix .PO) ready to be translated. The program understands enough about C, and about other languages (see box: xgettext: Supported Languages), to understand the syntax of a function call, and differentiate it from variables and comments.

```
$ xgettext -d lm helloworld.c
$ tail -n 3 lm.po
#: helloworld.c:5
```

xgettext: Supported Languages

C, C++, ObjectiveC	awk
PO	YCP
Python	Tcl
Lisp, EmacsLisp	RST
librep	Glade
Java	

Locale Categories

A category defines a set of data, and every supported language has its own set of data. The category might define the way to impart particular information: numbers over 1000 might be separated by commas or dots, for example, or the date might be written day-month-year or month-day-year. This information is not related to the language as such, which is why the term 'locale' is used, constituting both language and cultural specifics. A directory is created for each category.

There are standard functions to format these locale strings. For example, *strfmon* and *strftime* format the text for money and time data, respectively.

Category	Meaning
LC_COLLATE	Order of string-collation
LC_CTYPE	How to define characters. Echoes of ctype.h as this also performs upper/lower case conversion
LC_MESSAGES	The translated text. The focus of this article
LC_MONETARY	Format and symbols for money
LC_NUMERIC	Format and symbols for numbers
LC_TIME	Format and symbols for time and date

```
msgid "Hello World!\n"
msgstr ""
```

As you can see, each piece of text has a marker ID and an equivalent string, ready for translating. This string can only hold a translation for one specific language, so this file becomes a *template*. Each translator takes a copy of it, and translates the text within it to his or her native tongue. Sometimes, this PO file is renamed to POT to differentiate between the template, and the language-specific catalog files.

Note that *xgettext* will search for the function name *gettext*. It does not understand enough of the C syntax (or that any language) to understand techniques like *#define _(str)*, given above. This doesn't preclude the use of such tricks however. There are two popular solutions. One is to specify the underscore as an additional keyword that will act in the same manner as if it were *gettext*.

```
$ xgettext -d lm -k_ helloworld.c
```

Alternatively, you could pre-process your C file (causing the macro to be expanded) before running *xgettext*.

```
$ xgettext -C -d lm <(gcc -E ➥
helloworld.c)
```

In this example we specify the -C flag , to indicate that the piped result is a C source file. Users of automake will have an easier life, since the Makefile will generate these files automatically.

You will also note that the file contains comments using the familiar hash symbol. These comments come in four flavors, and are determined by the character immediately following the hash, as seen in Table 1.

The *xgettext* program can also add comments into the PO file when, for example, it believes the strings may be used for special formatting. The PO file also contains a header to indicate the revision date of the file, and the translator that last edited it.

Having now gotten this template file, we need to create a catalog for a foreign language. Like French.

Tour De France

We start by making a simple copy of the template file, and adding the appropriate French words to each *msgstr*.

```
msgid "Hello World!\n"
msgstr "Bonjour, le monde"
```

We can add the string(s) by either modifying the file directly, or using one of the many tools available. Translators using the Emacs editor have an advantage here, since they may use PO mode. For those who favor a GUI, the program *poeditor* can also be used.

To be used by our *Hello World* program, this text file needs to be converted into a machine-friendly, binary, format. The program that does this is called *msgfmt* and creates a file (ending in *.mo* instead of *.po*) that is more optimal for accessing arbitrary strings. It is not only trivial to use, but includes error checking which highlights the deliberate mistake above. Did you spot it? See Listing 1.

The first warning simply reminds us that we haven't changed the header information yet. We can fix that by amending the line to use the appropriate characterization.

```
"Content-Type: text/plain; ➥
charset=ISO-8859-1\n"
```

To determine an appropriate code you can refer to the box: ISO 8859, or [1] for a more detailed analysis. This information is of more use to translators than programmers. As is the extensive functionality provided by [2].

The error itself is easily fixed, and in larger programs, more difficult to spot by humans. It can also check the strings for the correct number (and type) of arguments using the *-c* option. We're now ready to test it!

Norwegian Wood

In order to convince our program to use an appropriate language dictionary, we need to add a couple of further lines of code to indicate that we're happy about using a locale. These are straight forward, and common to all such programs.

```
#include <locale.h>
...
char *pPackage = "lm";
char *pDirectory = "locale";
...
setlocale (LC_ALL, "");
bindtextdomain (pPackage, ➥
pDirectory);
textdomain (pPackage);
```

The *bindtextdomain* function indicates the local root directory of our translated catalog files, while *textdomain* requires us to specify the name of our *package*, or program. Ours is called 'lm', since we've created an *lm.mo* catalog. Note that if you specify a relative path for the *locale*directory, be careful not to change directory, as this path would then become unreachable.

While in our local root directory, we must create a *locale* directory, and copy our *lm.mo* to the appropriate place in the tree. That place being,

```
$ mkdir -p locale/fr/LC_MESSAGES
$ cp lm.mo locale/fr/LC_MESSAGES
```

Since the package is called 'lm.mo' in every language, we use the directory name to distinguish between a French *lm.mo* and a German *lm.mo*. This name is determined by the conventional language codes, as detailed at [3]. The directory named LC_MESSAGES is needed because of the wide variety of different locale information that might be present. There can also be directories to indicate the format of the date, and how to represent numbers. See box: Locale Categories for a full list.

Now you can run your program (without having to recompile), using a French locale, and witness the result.

```
$ LANG=fr_FR ./hello
Bonjour, le monde
```

For a more permanent change of locale, you must export the LANG environmental variable in the usual way. For example,

```
$ export LANG=fr_FR
$ ./hello
Bonjour, le monde
```

If you're on an exclusively English system this may not work, due to the fact there is no French locale on your system (other potential problems are covered in [4]). The */etc/locale.gen* file will indicate which locales have been generated for your machine, whereas the file */usr/share/i18n/SUPPORTED* will indicate which ones *can* be installed (along with

Table 1:Hash symbols

Character	Comment type	Notes
. (period)	Automatic	Should not be touched
: (colon)	Reference	The file & line number of the string
, (comma)	Flag	To indicate the translation is 'fuzzy', for example
(whitespace)	Translator	As entered by a human

Listing 1: Finding an error

```
01 $ msgfmt  lm.po
02 msgfmt: lm.po: warning: Charset "CHARSET" is not a portable encoding
   name.
03         Message conversion to user's charset might not work.
04 lm.po:19: `msgid' and `msgstr' entries do not both end with '\n'
05 msgfmt: found 1 fatal error
```

their appropriate ISO-8859 sets). Generating a French locale can be done easily with,

```
$ su
# you must be root to do this
Password:
# echo "fr_FR ISO-8859-1" >> ➥
/etc/locale.gen
# locale-gen
Generating locales...
  fr_FR.ISO-8859-1... done
Generation complete.
```

Debian users can also use *dpkg-reconfigure locales*.

You can test this using your own program, or (if you think the bug belongs to Hello World!) one of the multi-lingual GNU tools, such as *rm*.

```
$ LANG=fr_FR rm this_wont_exist
rm: Ne peut enlever `this_wont➥
_exist': Aucun fichier ou ➥
répertoire de ce type
```

To make your dictionary available to others, you should install it into the global repository of *.mo* files at */usr/share/locale/* (or the location specified by the environment variable, TEXTDOMAIN DIR). This directory uses the same hierarchy given above. Installing your text here (which also requires superuser privileges) means your code no longer needs to specify a directory to the *bindtextdomain* function, and you can replace the directory name with NULL.

Having now understood the technical process behind multi-lingual software, let us review some of the finer details we need to consider when programming.

Spanish Eyes

Most developers have a method for dealing with strings, like their favorite string library, for example. They also have their own methods for building strings dynamically, either to add plurals, or build large sentences from component parts (like the verbal Lego of automated train announcements). We shall now cover a number of these methods, highlighting the problems (and solutions) involved.

```
printf("Deleting %d file%s", ➥
iNum, iNum==1?"":"s");
```

Above is a common example to create a plural. The case of 'one file' requires a singular noun, whereas everything else uses the plural, *files*. That's in English! Not all languages follow this pattern. The case of 'zero files' might not be plural (as in French), or there could be separate words for zero, one and two (such as those in the Baltic family). To compensate for this, a separate function, *ngettext*, is available which takes two string ID's (one for singular, and one for plural) and a number. The number is then used to determine which version of that string should be used in translation.

```
printf( ngettext("Deleting %d ➥
file", "Deleting %d files", ➥
iNum), iNum);
```

Upon seeing the *ngettext* marker, the *xgettext* program will generate two string IDs in the *.PO* file, ready for the translator, along with a special *c-format* comment, which we'll come to shortly.

```
#: helloworld.c:32
#, c-format
msgid "Deleting %d file"
msgid_plural "Deleting %d files"
msgstr[0] ""
msgstr[1] ""
```

Not all the problems are solved by *ngettext* though. At some point you will come across problems that occur when we use two or more arguments in a *printf*, because the word order is imperative. Even in a simple (English) program, a mismatched %d and %s can cause *printf* to core dump. After translating a simple phrase, such as "There are %d files named %s", it is not unreasonable for the resultant text to appear as "With the name %s, there are %d files". What's more, since we (as programmers) do not know about every other possible translation, it is not something we can prevent. More subtle problems can occur with phrases like "Copying file from %s to %s".

There are two methods of resolving the word order problem. The first requires that the translator modify the wording so that the arguments always appear in the right order. The *msgfmt* command can then be called using the *-c* option, so that it will perform checks on the *.PO* file. This option actually performs three separate checks. They are, *format* (the one we need in this instance), *header* (the presence and contents of the header) and *domain* (checking for problems with the domain directives).

The second solution places the onus on the programmer, and is preferred. In this case, the format string must be amended to describe the order of the parameters. So, using our copy file example above, this would give us,

```
printf( gettext ("Copying file ➥
from %1$s to %2$s"), ➥
pSrc, pDest);
```

The special format specifiers, *%1$s* and *%2$s*, are handled by the *printf* code in *glibc*. Non-GNU variants may not be so feature-full.

Having highlighted the word order problem, you should now be aware that constructing strings at run-time is a bad idea. The solutions we have available to us can only work when the entire string is given to the translator. Splitting text up into sections and using *strcat* (or similar) should be avoided at all costs, since the translator has no understanding of the ordering (or the ability to change it), or the meaning of the sentence. Each string contained in the catalog must make sense when presented on its own.

ISO 8859

ISO	Characterization
ISO 8859-1	Western, or west European
ISO 8859-2	Central European, or east European
ISO 8859-3	South European, or Maltese (and Esperanto)
ISO 8859-4	North European
ISO 8859-5	Eastern European, Cyrillic alphabets like Russian
ISO 8859-6	Arabic
ISO 8859-7	Greek
ISO 8859-8	Hebrew
ISO 8859-9	Turkish
ISO 8859-10	Nordic (Sámi, Inuit, Icelandic)
ISO 8859-11	Thai
ISO 8859-12	(was Celtic, but withdrawn)
ISO 8859-13	The Baltic Rim
ISO 8859-14	Celtic
ISO 8859-15	Euro
ISO 8859-16	South eastern European (incorporates euro symbol)

Unicode

All the examples in this article use ASCII characters. This covers most western languages, but neglects those character sets requiring two bytes, such as Chinese. In order to support them fully, we need to work in Unicode. This involves a much larger quantity of work, as the basic *char* type can not be used, and is instead replaced by *wchar_t*. Also, many of the well-known functions (like sprintf) need to be adapted to use their equivalent *wide* versions, like *swprintf*.

```
/* Don't code like this!! */
strcpy("Copying file from ");
strcat(pSrc);
strcat(" to ");
strcat(pDest);
```

In some applications, the most difficult word to translate is 'the'! English has only one word for the definite article, 'the'. French, German, Spanish, and many others don't. Depending on the language, they may have special versions for masculine, feminine, neuter and plural. The same is true of the indefinite article, 'a'. Normally, these words will be included as part of the standard translation. By now you should have learnt that building strings dynamically is not a good idea. In some cases it can be very tempting to cut down on the quantity of translations required as in Listing 2.

We should modify this so that the strings read '*a* directory' and '*a* file', so the translated versions will work regardless of gender. However, you might argue, if we also had a portion of the program that produced a short version of the file listing, we would be doubling the work for the translator! For instance, in Listing 3.

That's true. We are doubling the work! However, this extra work is minimal. Especially compared to the programmer hassle that might otherwise be involved, or the cringe-inducing gender misuse when the wrong version of 'the' is prepended to the words.

China Girl

The last implementation problem we shall mention involves aesthetics. This refers to the screen layout, the menus of a GUI, and the use of tab stops. Although your program may look nicely formatted in English, as soon as any of the words change, your pre-determined layout will break. German words, for example, are on average 50% longer than their English equivalents. You have two choices. Either ignore word length, or code around it.

Most (if not all) command line utilities are unconcerned with special formatting. The information is functional and uniform, making it suitable for parsing by scripts. GUI software may explicitly place text in two columns, at X1 and X2, in order to appeal to the end user. There's nothing wrong with wanting to appeal to the end user! Unfortunately, when running under a different locale, the text in the left column may overrun the text in the right.

To avoid this problem you will need to write some more code. This might involve adjusting the position of the right hand column, perhaps by calculating the longest piece of text in the left, or you might need to word-wrap everything. It might involve scrolling the text within the visible window (like XMMS). It might simply chop all characters that overrun, and ask the translator for shorter versions. The solution you employ will vary according to the amount of work you, and your translators, are willing to do. Only applications that sell on their presentation abilities (like games) should consider this a necessity.

Vienna

As software develops, more and more strings will be added to the program. Retranslating the whole program every time is obviously wasted effort. So instead, we should use the *msgmerge* tool. This takes the original language template (the .PO file, that's often renamed to .POT) without any translations, and the newest language-specific catalog to build a new .PO. This new file contains all the original translations, combined with the new, as yet untranslated, strings.

```
$ msgmerge old_po_file.pot ➥
current_language_po.po >➥
new_language_po.po
```

Metropolis

With the *gettext* package, we can create truly multi-lingual software, even if we can't speak any of the languages in question. Using separate language catalogs allows the translation work to be distributed amongst those who can speak different tongues, without having to recompile the code. This makes it a fully data-driven, distributed, piece of development work.

So with that thought I bid you all a fond farewell. Au revoir. Auf Wiedersehen. Adiós and Arrivederci! ■

Listing 2: Smaller translations

```
01 if ( mygetfiletype(szFilename) == DIRECTORY)
02         pFiletype = gettext ("directory");
03 if ( mygetfiletype(szFilename) == FILE)
04         pFiletype = gettext ("file");
05 printf (gettext ("%1$s is a %2$s"), szFilename, pFiletype);
```

Listing 3: Doubling the work

```
01 if ( mygetfiletype(szFilename) == DIRECTORY)
02         pFiletype = gettext ("directory");      /* same strings as
   before - does this mean less work? */
03 if ( mygetfiletype(szFilename) == FILE)
04         pFiletype = gettext ("file");
05 printf("%s : %s", szFilename, pFiletype);       /* no translation
   required here */
```

INFO

[1] ISO8859 Alphabet Soup: *http://wwwwbs.cs.tu-belin.de/user/czyborra/charsets/*

[2] Data on languages: *http://www.eki.ee/letter/*

[3] Language codes: *http://www.loc.gov/standards/iso639-2/langcodes.html*

[4] FAQ for GNU gettext: *http://www.haible.de/bruno/gettext-FAQ.html#integrating_noop*