9

Internationalization with gawk

In this chapter:

- *Internationalization and Localization*
- *GNU gettext*
- *Internationalizing awk Programs*
- *Translating awk Programs*
- *A Simple Internationalization Example*
- *gawk Can Speak Your Language*

Once upon a time, computer makers wrote software that worked only in English. Eventually, hardware and software vendors noticed that if their systems worked in the native languages of non-English-speaking countries, they were able to sell more systems. As a result, internationalization and localization of programs and software systems became a common practice.

Until recently, the ability to provide internationalization was largely restricted to programs written in C and C++. This chapter describes the underlying library *gawk* uses for internationalization, as well as how *gawk* makes internationalization features available at the *awk* program level. Having internationalization available at the *awk* level gives software developers additional flexibility—they are no longer required to write in C when internationalization is a requirement.

Internationalization and Localization

Internationalization means writing (or modifying) a program once, in such a way that it can use multiple languages without requiring further source-code changes. *Localization* means providing the data necessary for an internationalized program to work in a particular language. Most typically, these terms refer to features such as the language used for printing error messages, the language used to read responses, and information related to how numerical and monetary values are printed and read.

9 October 2001 01:42

GNU gettext

The facilities in GNU `gettext` focus on messages; strings printed by a program, either directly or via formatting with `printf` or `sprintf`.*

When using GNU `gettext`, each application has its own *text domain*. This is a unique name, such as `kpilot` or `gawk`, that identifies the application. A complete application may have multiple components—programs written in C or C++, as well as scripts written in *sh* or *awk*. All of the components use the same text domain.

To make the discussion concrete, assume we're writing an application named *guide*. Internationalization consists of the following steps, in this order:

1. The programmer goes through the source for all of *guide*'s components and marks each string that is a candidate for translation. For example, `"`-F': option required"` is a good candidate for translation. A table with strings of option names is not (e.g., *gawk*'s *––profile* option should remain the same, no matter what the local language).
2. The programmer indicates the application's text domain (`"guide"`) to the `gettext` library, by calling the `textdomain` function.
3. Messages from the application are extracted from the source code and collected into a portable object file (*guide.po*), which lists the strings and their translations. The translations are initially empty. The original (usually English) messages serve as the key for lookup of the translations.
4. For each language with a translator, *guide.po* is copied and translations are created and shipped with the application.
5. Each language's *.po* file is converted into a binary message object (*.mo*) file. A message object file contains the original messages and their translations in a binary format that allows fast lookup of translations at runtime.
6. When *guide* is built and installed, the binary translation files are installed in a standard place.
7. For testing and development, it is possible to tell `gettext` to use *.mo* files in a different directory than the standard one by using the `bindtextdomain` function.
8. At runtime, *guide* looks up each string via a call to `gettext`. The returned string is the translated string if available, or the original string if not.

---

* For some operating systems, the *gawk* port doesn't support GNU `gettext`. This applies most notably to the PC operating systems. As such, these features are not available if you are using one of those operating systems. Sorry.

9. If necessary, it is possible to access messages from a different text domain than the one belonging to the application, without having to switch the application's default text domain back and forth.

In C (or C++), the string marking and dynamic translation lookup are accomplished by wrapping each string in a call to `gettext`:

```
printf(gettext("Don't Panic!\n"));
```

The tools that extract messages from source code pull out all strings enclosed in calls to `gettext`.

The GNU `gettext` developers, recognizing that typing `gettext` over and over again is both painful and ugly to look at, use the macro `_` (an underscore) to make things easier:

```
/* In the standard header file: */
#define _(str) gettext(str)

/* In the program text: */
printf(_("Don't Panic!\n"));
```

This reduces the typing overhead to just three extra characters per string and is considerably easier to read as well. There are locale *categories* for different types of locale-related information. The defined locale categories that `gettext` knows about are:

`LC_MESSAGES`
: Text messages. This is the default category for `gettext` operations, but it is possible to supply a different one explicitly, if necessary. (It is almost never necessary to supply a different category.)

`LC_COLLATE`
: Text-collation information; i.e., how different characters and/or groups of characters sort in a given language.

`LC_CTYPE`
: Character-type information (alphabetic, digit, upper- or lowercase, and so on). This information is accessed via the POSIX character classes in regular expressions, such as `/[[:alnum:]]/` (see the section "Regular Expression Operators" in Chapter 2, *Regular Expressions*).

`LC_MONETARY`
: Monetary information, such as the currency symbol, and whether the symbol goes before or after a number.

`LC_NUMERIC`
: Numeric information, such as which characters to use for the decimal point and the thousands separator.*

`LC_RESPONSE`
: Response information, such as how "yes" and "no" appear in the local language, and possibly other information as well.

`LC_TIME`
: Time- and date-related information, such as 12- or 24-hour clock, month printed before or after day in a date, local month abbreviations, and so on.

`LC_ALL`
: All of the above. (Not too useful in the context of `gettext`.)

Internationalizing awk Programs

gawk provides the following variables and functions for internationalization:

`TEXTDOMAIN`
: This variable indicates the application's text domain. For compatibility with GNU `gettext`, the default value is `"messages"`.

`_"your message here"`
: String constants marked with a leading underscore are candidates for translation at runtime. String constants without a leading underscore are not translated.

`dcgettext(`*`string`*`[, `*`domain`*`[, `*`category`*`]])`
: This built-in function returns the translation of *string* in text domain *domain* for locale category *category*. The default value for *domain* is the current value of `TEXTDOMAIN`. The default value for *category* is `"LC_MESSAGES"`.

  If you supply a value for *category*, it must be a string equal to one of the known locale categories described in the previous section. You must also supply a text domain. Use `TEXTDOMAIN` if you want to use the current domain.

The order of arguments to the *awk* version of the `dcgettext` function is purposely different from the order for the C version. The *awk* version's order was chosen to be simple and to allow for reasonable *awk*-style default arguments.

* Americans use a comma every three decimal places and a period for the decimal point, while many Europeans do exactly the opposite: `1,234.56` versus `1.234,56`.

`bindtextdomain(`*`directory`*`[, `*`domain`*`])`

This built-in function allows you to specify the directory in which `gettext` looks for *.mo* files, in case they will not or cannot be placed in the standard locations (e.g., during testing). It returns the directory in which *domain* is "bound."

The default *domain* is the value of `TEXTDOMAIN`. If *directory* is the null string (`""`), then `bindtextdomain` returns the current binding for the given *domain*.

To use these facilities in your *awk* program, follow the steps outlined in the previous section, like so:

1. Set the variable `TEXTDOMAIN` to the text domain of your program. This is best done in a `BEGIN` rule (see the section "The BEGIN and END Special Patterns" in Chapter 6, *Patterns, Actions, and Variables*), or it can also be done via the *–v* command-line option (see the section "Command-Line Options" in Chapter 11, *Running awk and gawk*):

   ```
   BEGIN {
       TEXTDOMAIN = "guide"
       ...
   }
   ```

2. Mark all translatable strings with a leading underscore (`_`) character. It *must* be adjacent to the opening quote of the string. For example:

   ```
   print _"hello, world"
   x = _"you goofed"
   printf(_"Number of users is %d\n", nusers)
   ```

3. If you are creating strings dynamically, you can still translate them, using the `dcgettext` built-in function:

   ```
   message = nusers " users logged in"
   message = dcgettext(message, "adminprog")
   print message
   ```

   Here, the call to `dcgettext` supplies a different text domain (`"adminprog"`) in which to find the message, but it uses the default `"LC_MESSAGES"` category.

4. During development, you might want to put the *.mo* file in a private directory for testing. This is done with the `bindtextdomain` built-in function:

   ```
   BEGIN {
       TEXTDOMAIN = "guide"   # our text domain
       if (Testing) {
           # where to find our files
           bindtextdomain("testdir")
           # joe is in charge of adminprog
           bindtextdomain("../joe/testdir", "adminprog")
       }
       ...
   }
   ```

See the section "A Simple Internationalization Example" later in this chapter for an example program showing the steps to create and use translations from *awk*.

Translating awk Programs

Once a program's translatable strings have been marked, they must be extracted to create the initial *.po* file. As part of translation, it is often helpful to rearrange the order in which arguments to `printf` are output.

gawk's *––gen–po* command-line option extracts the messages and is discussed next. After that, `printf`'s ability to rearrange the order for `printf` arguments at runtime is covered.

Extracting Marked Strings

Once your *awk* program is working, and all the strings have been marked and you've set (and perhaps bound) the text domain, it is time to produce translations. First, use the *––gen–po* command-line option to create the initial *.po* file:

```
$ gawk --gen-po -f guide.awk > guide.po
```

When run with *––gen–po*, *gawk* does not execute your program. Instead, it parses it as usual and prints all marked strings to standard output in the format of a GNU `gettext` Portable Object file. Also included in the output are any constant strings that appear as the first argument to `dcgettext`.* See the section "A Simple Internationalization Example" later in this chapter for the full list of steps to go through to create and test translations for *guide*.

Rearranging printf Arguments

Format strings for `printf` and `sprintf` (see the section "Using printf Statements for Fancier Printing" in Chapter 4, *Printing Output*) present a special problem for translation. Consider the following:†

```
printf(_"String `%s' has %d characters\n", string, length(string)))
```

A possible German translation for this might be:

```
"%d Zeichen lang ist die Zeichenkette `%s'\n"
```

The problem should be obvious: the order of the format specifications is different from the original! Even though `gettext` can return the translated string at runtime, it cannot change the argument order in the call to `printf`.

---

* Eventually, the *xgettext* utility that comes with GNU `gettext` will be taught to automatically run `gawk --gen-po` for *.awk* files, freeing the translator from having to do it manually.

† This example is borrowed from the GNU `gettext` manual.

To solve this problem, `printf` format specificiers may have an additional optional element, which we call a *positional specifier*. For example:

```
"%2$d Zeichen lang ist die Zeichenkette '%1$s'\n"
```

Here, the positional specifier consists of an integer count, which indicates which argument to use, and a `$`. Counts are one-based, and the format string itself is *not* included. Thus, in the following example, `string` is the first argument and `length(string)` is the second:

```
$ gawk 'BEGIN {
>     string = "Dont Panic"
>     printf _"%2$d characters live in \"%1$s\"\n",
>                          string, length(string)
> }'
10 characters live in "Dont Panic"
```

If present, positional specifiers come first in the format specification, before the flags, the field width, and/or the precision.

Positional specifiers can be used with the dynamic field width and precision capability:

```
$ gawk 'BEGIN {
>    printf("%*.*s\n", 10, 20, "hello")
>    printf("%3$*2$.*1$s\n", 20, 10, "hello")
> }'
hello
hello
```

When using `*` with a positional specifier, the `*` comes first, then the integer position, and then the `$`. This is somewhat counterintuitive.

gawk does not allow you to mix regular format specifiers and those with positional specifiers in the same string:

```
$ gawk 'BEGIN { printf _"%d %3$s\n", 1, 2, "hi" }'
gawk: cmd. line:1: fatal: must use 'count$' on all formats or none
```

There are some pathological cases that *gawk* may fail to diagnose. In such cases, the output may not be what you expect. It's still a bad idea to try mixing them, even if *gawk* doesn't detect it.

Although positional specifiers can be used directly in *awk* programs, their primary purpose is to help in producing correct translations of format strings into languages different from the one in which the program is first written.

awk Portability Issues

gawk's internationalization features were purposely chosen to have as little impact as possible on the portability of *awk* programs that use them to other versions of *awk*. Consider this program:

```
BEGIN {
    TEXTDOMAIN = "guide"
    if (Test_Guide)   # set with -v
        bindtextdomain("/test/guide/messages")
    print _"don't panic!"
}
```

As written, it won't work on other versions of *awk*. However, it is actually almost portable, requiring very little change:

- Assignments to `TEXTDOMAIN` won't have any effect, since `TEXTDOMAIN` is not special in other *awk* implementations.
- Non-GNU versions of *awk* treat marked strings as the concatenation of a variable named `_` with the string following it.* Typically, the variable `_` has the null string (`""`) as its value, leaving the original string constant as the result.
- By defining "dummy" functions to replace `dcgettext` and `bindtextdomain`, the *awk* program can be made to run, but all the messages are output in the original language. For example:

  ```
  function bindtextdomain(dir, domain)
  {
      return dir
  }

  function dcgettext(string, domain, category)
  {
      return string
  }
  ```

- The use of positional specifications in `printf` or `sprintf` is *not* portable. To support `gettext` at the C level, many systems' C versions of `sprintf` do support positional specifiers. But it works only if enough arguments are supplied in the function call. Many versions of *awk* pass `printf` formats and arguments unchanged to the underlying C library version of `sprintf`, but only one format and argument at a time. What happens if a positional specification is used is

* This is good fodder for an "Obfuscated *awk*" contest.

anybody's guess. However, since the positional specifications are primarily for use in *translated* format strings, and since non-GNU *awk*s never retrieve the translated string, this should not be a problem in practice.

A Simple Internationalization Example

Now let's look at a step-by-step example of how to internationalize and localize a simple *awk* program, using *guide.awk* as our original source:

```
BEGIN {
    TEXTDOMAIN = "guide"
    bindtextdomain(".")  # for testing
    print _"Don't Panic"
    print _"The Answer Is", 42
    print "Pardon me, Zaphod who?"
}
```

Run `gawk --gen-po` to create the *.po* file:

```
$ gawk --gen-po -f guide.awk > guide.po
```

This produces:

```
#: guide.awk:4
msgid "Don't Panic"
msgstr ""

#: guide.awk:5
msgid "The Answer Is"
msgstr ""
```

This original portable object file is saved and reused for each language into which the application is translated. The `msgid` is the original string and the `msgstr` is the translation.

Strings not marked with a leading underscore do not appear in the *guide.po* file.

Next, the messages must be translated. Here is a translation to a hypothetical dialect of English, called "Mellow":*

```
$ cp guide.po guide-mellow.po
Add translations to guide-mellow.po ...
```

* Perhaps it would be better if it were called "Hippy." Ah, well.

Following are the translations:

```
#: guide.awk:4
msgid "Don't Panic"
msgstr "Hey man, relax!"

#: guide.awk:5
msgid "The Answer Is"
msgstr "Like, the scoop is"
```

The next step is to make the directory to hold the binary message object file and then to create the *guide.mo* file. The directory layout shown here is standard for GNU `gettext` on GNU/Linux systems. Other versions of `gettext` may use a different layout:

```
$ mkdir en_US en_US/LC_MESSAGES
```

The *msgfmt* utility does the conversion from human-readable *.po* file to machine-readable *.mo* file. By default, *msgfmt* creates a file named *messages*. This file must be renamed and placed in the proper directory so that *gawk* can find it:

```
$ msgfmt guide-mellow.po
$ mv messages en_US/LC_MESSAGES/guide.mo
```

Finally, we run the program to test it:

```
$ gawk -f guide.awk
Hey man, relax!
Like, the scoop is 42
Pardon me, Zaphod who?
```

If the two replacement functions for `dcgettext` and `bindtextdomain` (see the section "awk Portability Issues" earlier in this chapter) are in a file named *libintl.awk*, then we can run *guide.awk* unchanged as follows:

```
$ gawk --posix -f guide.awk -f libintl.awk
Don't Panic
The Answer Is 42
Pardon me, Zaphod who?
```

gawk Can Speak Your Language

As of Version 3.1, *gawk* itself has been internationalized using the GNU `gettext` package. (GNU `gettext` is described in complete detail in *GNU gettext tools*.) As of this writing, the latest version of GNU `gettext` is Version 0.10.37 (*ftp://gnudist.gnu.org/gnu/gettext/gettext-0.10.37.tar.gz*).

If a translation of *gawk*'s messages exists, then *gawk* produces usage messages, warnings, and fatal errors in the local language.

On systems that do not use Version 2 (or later) of the GNU C library, you should configure *gawk* with the *--with-included-gettext* option before compiling and installing it. See the section "Additional Configuration Options" in Appendix B, *Installing gawk*, for more information.