NAME

Locale::Messages − Gettext Like Message Retrieval

SYNOPSIS

```
 use Locale::Messages (:locale_h :libintl_h);

 gettext $msgid;
 dgettext $textdomain, $msgid;
 dcgettext $textdomain, $msgid, LC_MESSAGES;
 ngettext $msgid, $msgid_plural, $count;
 dngettext $textdomain, $msgid, $msgid_plural, $count;
 dcngettext $textdomain, $msgid, $msgid_plural, $count, LC_MESSAGES;
 pgettext $msgctxt, $msgid;
 dpgettext $textdomain, $msgctxt, $msgid;
 dcpgettext $textdomain, $msgctxt, $msgid, LC_MESSAGES;
 npgettext $msgctxt, $msgid, $msgid_plural, $count;
 dnpgettext $textdomain, $msgctxt, $msgid, $msgid_plural, $count;
 dcnpgettext $textdomain, $msgctxt, $msgid, $msgid_plural, $count, LC_MESSAGES;
 textdomain $textdomain;
 bindtextdomain $textdomain, $directory;
 bind_textdomain_codeset $textdomain, $encoding;
 bind_textdomain_filter $textdomain, \&filter, $data;
 turn_utf_8_on ($variable);
 turn_utf_8_off ($variable);
 nl_putenv ('OUTPUT_CHARSET=koi8-r');
 my $category = LC_CTYPE;
 my $category = LC_NUMERIC;
 my $category = LC_TIME;
 my $category = LC_COLLATE;
 my $category = LC_MONETARY;
 my $category = LC_MESSAGES;
 my $category = LC_ALL;
```

DESCRIPTION

The module **Locale::Messages** is a wrapper around the interface to message translation according to the Uniforum approach that is for example used in GNU gettext and Sun's Solaris. It is intended to allow **Locale::Messages**(3) to switch between different implementations of the lower level libraries but this is not yet implemented.

Normally you should not use this module directly, but the high level interface **Locale::TextDomain**(3) that provides a much simpler interface. This description is therefore deliberately kept brief. Please refer to the GNU gettext documentation available at <http://www.gnu.org/manual/gettext/> for in-depth and background information on the topic.

The lower level module **Locale::gettext_pp**(3) provides the Perl implementation of **gettext()** and related functions.

FUNCTIONS

The module exports by default nothing. Every function has to be imported explicitly or via an export tag ("EXPORT TAGS").

gettext MSGID

Returns the translation for **MSGID**. Example:

```
    print gettext "Hello World!\n";
```

If no translation can be found, the unmodified **MSGID** is returned, i. e. the function can *never* fail, and will *never* mess up your original message.

Note for Perl 5.6 and later: The returned string will *always* have the UTF−8 flag off by default. See the documentation for function **bind_textdomain_filter()** for a way to change this behavior.

One common mistake is this:

```
    print gettext "Hello $name!";
```

Perl will interpolate the variable `$name` *before* the function will see the string. Unless the corresponding message catalog contains a message ‘‘Hello Tom!’’, ‘‘Hello Dick!’’ or ‘‘Hello Harry!’’, no translation will be found.

Using **printf()** and friends has its own problems:

```
    print sprintf (gettext ("This is the %s %s."), $color, $thing);
```

(The example is stupid because neither color nor thing will get translated here ...).

In English the adjective (the color) will precede the noun, many other languages (for example French or Italian) differ here. The translator of the message may therefore have a hard time to find a translation that will still work and not sound stupid in the target language. Many C implementations of **printf()** allow one to change the order of the arguments, and a French translator could then say:

```
    "C'est le %2$s %1$s."
```

Perl **printf()** implements this feature as of version 5.8 or better. Consequently you can only use it, if you are sure that your software will run with Perl 5.8 or a later version.

Another disadvantage of using **printf()** is its cryptic syntax (maybe not for you but translators of your software may have their own opinion).

See the description of the function `__x()` in **Locale::TextDomain**(3) for a much better way to get around this problem.

Non-ASCII message ids ...

You should note that the function (and all other similar functions in this module) does a bytewise comparison of the **MSGID** for the lookup in the translation catalog, no matter whether obscure utf−8 flags are set on it, whether the string looks like utf−8, whether the **utf8**(3pm) pragma is used, or whatever other weird method past or future **perl**(1) versions invent for guessing character sets of strings.

Using other than us-ascii characters in Perl source code is a call for trouble, a compatibility nightmare. Furthermore, GNU gettext only lately introduced support for non-ascii character sets in sources, and support for this feature may not be available everywhere. If you absolutely want to use **MSGID**s in non-ascii character sets, it is wise to choose utf−8. This will minimize the risk that **perl**(1) itself will mess with the strings, and it will also be a guaranty that you can later translate your project into arbitrary target languages.

Other character sets can theoretically work. Yet, using another character set in the Perl source code than the one used in your message catalogs will **never** work, since the lookup is done bytewise, and all strings with non-ascii characters will not be found.

Even if you have solved all these problems, there is still one show stopper left: The gettext runtime API lacks a possibility to specify the character set of the source code (including the original strings). Consequently − in absence of a hint for the input encoding − strings without a translation are not subject to output character set conversion. In other words: If the (non-determinable) output character set differs from the character set used in the source code, output can be a mixture of two character sets. There is no point in trying to address this problem in the pure Perl version of the gettext functions. because breaking compatibility between the Perl and the C version is a price too high to pay.

This all boils down to: Only use ASCII characters in your translatable strings!

dgettext TEXTDOMAIN, MSGID

Like **gettext()**, but retrieves the message for the specified **TEXTDOMAIN** instead of the default domain. In case you wonder what a textdomain is, you should really read on with

Locale::TextDomain (3).

dcgettext TEXTDOMAIN, MSGID, CATEGORY

Like **dgettext()** but retrieves the message from the specified **CATEGORY** instead of the default category `LC_MESSAGES`.

ngettext MSGID, MSGID_PLURAL, COUNT

Retrieves the correct translation for **COUNT** items. In legacy software you will often find something like:

```
print "$count file(s) deleted.\n";
```

or

```
printf "$count file%s deleted.\n", $count == 1 ? '' : 's';
```

The first example looks awkward, the second will only work in English and languages with similar plural rules. Before **ngettext()** was introduced, the best practice for internationalized programs was:

```
if ($count == 1) {
    print gettext "One file deleted.\n";
} else {
    printf gettext "%d files deleted.\n";
}
```

This is a nuisance for the programmer and often still not sufficient for an adequate translation. Many languages have completely different ideas on numerals. Some (French, Italian, ...) treat 0 and 1 alike, others make no distinction at all (Japanese, Korean, Chinese, ...), others have two or more plural forms (Russian, Latvian, Czech, Polish, ...). The solution is:

```
printf (ngettext ("One file deleted.\n",
                  "%d files deleted.\n",
                  $count), # argument to ngettext!
        $count);          # argument to printf!
```

In English, or if no translation can be found, the first argument (**MSGID**) is picked if `$count` is one, the second one otherwise. For other languages, the correct plural form (of 1, 2, 3, 4, ...) is automatically picked, too. You don't have to know anything about the plural rules in the target language, **ngettext()** will take care of that.

This is most of the time sufficient but you will have to prove your creativity in cases like

```
printf "%d file(s) deleted, and %d file(s) created.\n";
```

dngettext TEXTDOMAIN, MSGID, MSGID_PLURAL, COUNT

Like **ngettext()** but retrieves the translation from the specified textdomain instead of the default domain.

dcngettext TEXTDOMAIN, MSGID, MSGID_PLURAL, COUNT, CATEGORY

Like **dngettext()** but retrieves the translation from the specified category, instead of the default category `LC_MESSAGES`.

pgettext MSGCTXT, MSGID

Returns the translation of MSGID, given the context of MSGCTXT.

Both items are used as a unique key into the message catalog.

This allows the translator to have two entries for words that may translate to different foreign words based on their context. For example, the word ‘‘View’’ may be a noun or a verb, which may be used in a menu as File−>View or View−>Source.

```
pgettext "Verb: To View", "View\n";
pgettext "Noun: A View", "View\n";
```

The above will both lookup different entries in the message catalog.

A typical usage are GUI programs. Imagine a program with a main menu and the notorious ‘‘Open’’ entry in the ‘‘File’’ menu. Now imagine, there is another menu entry Preferences−>Advanced−>Policy where you have a choice between the alternatives ‘‘Open’’ and ‘‘Closed’’. In English, ‘‘Open’’ is the adequate text at both places. In other languages, it is very likely that you need two different translations. Therefore, you would now write:

```
pgettext "File|", "Open";
pgettext "Preferences|Advanced|Policy", "Open";
```

In English, or if no translation can be found, the second argument (MSGID) is returned.

The function was introduced with libintl-perl version 1.17.

dpgettext TEXTDOMAIN, MSGCTXT, MSGID

Like **pgettext()**, but retrieves the message for the specified **TEXTDOMAIN** instead of the default domain.

The function was introduced with libintl-perl version 1.17.

dcpgettext TEXTDOMAIN, MSGCTXT, MSGID, CATEGORY

Like **dpgettext()** but retrieves the message from the specified **CATEGORY** instead of the default category `LC_MESSAGES`.

The function was introduced with libintl-perl version 1.17.

npgettext MSGCTXT, MSGID, MSGID_PLURAL, COUNT

Like **ngettext()** with the addition of context as in **pgettext()**.

In English, or if no translation can be found, the second argument (MSGID) is picked if `$count` is one, the third one otherwise.

The function was introduced with libintl-perl version 1.17.

dnpgettext TEXTDOMAIN, MSGCTXT, MSGID, MSGID_PLURAL, COUNT

Like **npgettext()** but retrieves the translation from the specified textdomain instead of the default domain.

The function was introduced with libintl-perl version 1.17.

dcnpgettext TEXTDOMAIN, MSGCTXT, MSGID, MSGID_PLURAL, COUNT, CATEGORY

Like **dnpgettext()** but retrieves the translation from the specified category, instead of the default category `LC_MESSAGES`.

The function was introduced with libintl-perl version 1.17.

textdomain TEXTDOMAIN

Sets the default textdomain (initially ’messages’).

bindtextdomain TEXTDOMAIN, DIRECTORY

Binds **TEXTDOMAIN** to **DIRECTORY**. Huh? An example:

```
bindtextdomain "my−package", "./mylocale";
```

Say, the selected locale (actually the selected locale for category `LC_MESSAGES`) of the program is ’fr_CH’, then the message catalog will be expected in *./mylocale/fr_CH/LC_MESSAGES/my−package.mo*.

bind_textdomain_codeset TEXTDOMAIN, ENCODING

Sets the output encoding for **TEXTDOMAIN** to **ENCODING**.

bind_textdomain_filter TEXTDOMAN, CODEREF, DATA
bind_textdomain_filter TEXTDOMAN, CODEREF

By default, Locale::Messages will turn the utf−8 flag of all returned messages off. If you want to change this behavior, you can pass a reference to a subroutine that does different things − for example turn the utf−8 flag on, or leave it untouched. The callback function will be called with **DATA** as the first, and the possibly translated string as the second argument. It should return the possibly modified string.

If you want an object method to be called, pass the object itself in the data parameter and write a wrapper function. Example:

```
sub wrapper {
    my ($string, $obj) = @_;

    $obj->filterMethod ($string);
}
my $obj = MyPackage->new;

bind_textdomain_filter ('mydomain', \&wrapper, $obj);
```

The function cannot fail and always returns a true value.

Attention: If you use the function for setting the utf−8 flag, it is **your** responsibility to ensure that the output is really utf−8. You should only use it, if you have set the environment variable **OUTPUT_CHARSET** to ‘‘utf−8’’. Additionally you should call**bind_textdomain_codeset()** with ‘‘utf−8’’ as the second argument.

Steven Haryanto has written a module **Locale::TextDomain::UTF8** (3pm) that addresses the same problem.

This function has been introduced in libintl-perl 1.16 and it is **not** part of the standard gettext API.

turn_utf_8_on VARIABLE

Returns VARIABLE but with the UTF−8 flag (only known in Perl >=5.6) guaranteed to be turned on. This function does not really fit into the module, but it is often handy nevertheless.

The flag does **not** mean that the string is in fact valid utf−8!

The function was introduced with libintl-perl version 1.16.

turn_utf_8_off VARIABLE

Returns VARIABLE but with the UTF−8 flag (only known in Perl >=5.6) guaranteed to be turned off. This function does not really fit into the module, but it is often handy nevertheless.

The function was introduced with libintl-perl version 1.07.

select_package PACKAGE

By default, **Locale::Messages** will try to load the XS version of the gettext implementation, i. e. **Locale::gettext_xs** (3) and will fall back to the pure Perl implementation **Locale::gettext_pp** (3). You can override this behavior by passing the string ‘‘gettext_pp’’ or ‘‘gettext_xs’’ to the function **select_package()**. Passing ‘‘gettext_pp’’ here, will prefer the pure Perl implementation.

You will normally want to use that in a BEGIN block of your main script.

The function was introduced with libintl-perl version 1.03 and is not part of the standard gettext API.

Beginning with version 1.22 you can pass other package names than ‘‘gettext_pp’’ or ‘‘gettext_xs’’ and use a completely different backend. It is the caller’s responsibility to make sure that the selected package offers the same interface as the two standard packages.

One package that offers that functionality is **Locale::gettext_dumb** (3pm).

nl_putenv ENVSPEC

Resembles the ANSI C **putenv** (3) function. The sole purpose of this function is to work around some ideosyncrasies in the environment processing of Windows systems. If you want to portably set or unset environment variables, use this function instead of directly manipulating `%ENV`.

The argument **ENVSPEC** may have three different forms.

LANGUAGE=fr_CH

This would set the environment variable `LANGUAGE` to ‘‘fr_CH’’.

LANGUAGE=

Normally, this will set the environment variable `LANGUAGE` to an empty string. Under Windows, however, the environment variable will be deleted instead (and is no longer present in `%ENV`). Since within libintl-perl empty environment variables are useless, consider this usage as deprecated.

LANGUAGE

This will delete the environment variable **LANGUAGE**. If you are familiar with the brain-damaged implementation of **putenv** (3) (resp. **_puten v()**) in the so-called standard C library of MS-Windows, you may suspect that this is an invalid argument. This is not the case! Passing a variable name not followed by an equal sign will always delete the variable, no matter which operating system you use.

The function returns true for success, and false for failure. Possible reasons for failure are an invalid syntax or − only under Windows − failure to allocate space for the new environment entry ($! will be set accordingly in this case).

Why all this hassle? The 32−bit versions of MS-DOS (currently Windows 95/98/ME/NT/2000/XP/CE/.NET) maintain two distinct blocks of environment variables per process. Which block is considered the ‘‘correct’’ environment is a compile-time option of the Perl interpreter. Unfortunately, if you have build the XS version **Locale::gettext_xs** (3) under Windows, the underlying library may use a different environment block, and changes you make to `%ENV` may not be visible to the library.

The function **nl_putenv()** is mostly a funny way of saying

```
LANGUAGE=some_value
```

but it does its best, to pass this information to the gettext library. Under other operating systems than Windows, it only operates on `%ENV`, under Windows it will call the C library function **_putenv()** (after doing some cleanup to its arguments), before manipulating `%ENV`.

Please note, that your `%ENV` is updated by **nl_putenv()** automatically.

The function has been introduced in libintl-perl version 1.10.

setlocale

Modifies and queries program’s locale, see the documentation for **setlocale()** in **POSIX** (3pm) instead.

On some systems, when using GNU gettext, a call from C to **setlocale()** is − with the help of the C preprocessor − really a call to **libintl_setlocale()**, which is in turn a wrapper around the system **setlocale** (3). Failure to call **libintl_setlocale()** may lead to certain malfunctions. On such systems, **Locale::Messages::setlocale()** will call the wrapper **libintl_setlocale()**. If you want to avoid problems, you should therefore always call the **setlocale()** implementation in **Locale::Messages** (3pm).

See <https://rt.cpan.org/Public/Bug/Display.html?id=83980> or <https://savannah.gnu.org/bugs/?38162>, and <https://savannah.gnu.org/bugs/?func=detailitem&item_id=44645> for a discussion of the problem.

The function has been introduced in libintl-perl version 1.24.

CONSTANTS

You can (maybe) get the same constants from **POSIX** (3); see there for a detailed description

LC_CTYPE
LC_NUMERIC

LC_TIME
LC_COLLATE
LC_MONETARY
LC_MESSAGES

This locale category was the reason that these constants from **POSIX** (3) were included here. Even if it was present in your systems C include file *locale.h*, it was not provided by **POSIX** (3). Perl 5.8 and later seems to export the constant if available, although it is not documented in **POSIX** (3).

Locale::Messages (3) makes an attempt to guess the value of this category for all systems, and assumes the arbitrary value 1729 otherwise.

LC_ALL

If you specify the category **LC_ALL** as the first argument to **POSIX::setlocale()**, *all* locale categories will be affected at once.

EXPORT TAGS

The module does not export anything unless explicitly requested. You can import groups of functions via two tags:

use Locale::Messages (':locale_h')

Imports the functions that are normally defined in the C include file *locale.h*:

gettext()
dgettext()
dcgettext()
ngettext()
dngettext()
dcngettext()
pgettext()
dpgettext()
dcpgettext()
npgettext()
dnpgettext()
dcnpgettext()
textdomain()
bindtextdomain()
bind_textdomain_codeset()

use Locale::Messages (':libintl_h')

Imports the locale category constants:

LC_CTYPE
LC_NUMERIC
LC_TIME
LC_COLLATE
LC_MONETARY
LC_MESSAGES
LC_ALL

OTHER EXPORTS

select_package PACKAGE

USAGE

A complete example:

```
    1: use Locale::Messages qw (:locale_h :libintl_h);
    2: use POSIX qw (setlocale);
    3: setlocale (LC_MESSAGES, '');
    4: textdomain ('my−package');
    5: bindtextdomain ('my−package' => '/usr/local/share/locale');
    6:
    7: print gettext ("Hello world!\n");
```

Step by step: Line 1 imports the necessary functions and constants. In line 3 we set the locale for category LC_MESSAGES to the default user settings. For C programs you will often read that LC_ALL is the best category here but this will also change the locale for LC_NUMERIC and many programs will not work reliably after changing that category in Perl; choose your own poison!

In line 4 we say that all messages (translations) without an explicit domain specification should be retrieved from the message catalog for the domain 'my−package'. Line 5 has the effect that the message catalog will be searched under the directory */usr/local/share/locale*.

If the user has selected the locale 'fr_CH', and if the file */usr/local/share/locale/fr_CH/LC_MESSAGES/my−package.mo* exists, and if it contains a GNU message object file with a translation for the string ‘‘Hello world!\n’’, then line 7 will print the French translation (for Switzerland CH) to STDOUT.

The documentation for GNU gettext explains how to extract translatable strings from your Perl files and how to create message catalogs.

Another less portable example: If your system uses the GNU libc you should be able to find various files with the name *libc.mo*, the message catalog for the library itself. If you have found these files under */usr/share/locale*, then you can try the following:

```
    use Locale::Messages qw (:locale_h :libintl_h);
    use POSIX qw (setlocale);

    setlocale LC_MESSAGES, "";
    textdomain "libc";

    # The following is actually not needed, since this is
    # one of the default search directories.
    bindtextdomain libc => '/usr/share/locale';
    bind_textdomain_codeset libc => 'iso−8859−1';

    print gettext ("No such file or directory");
```

See **Locale::TextDomain** (3) for much simpler ways.

AUTHOR

SEE ALSO

Locale::TextDomain (3pm), **Locale::gettext_pp** (3pm), **Encode** (3pm), **Locale::TextDomain::UTF8** (3pm), **perllocale** (3pm), **POSIX** (3pm), **perl** (1), **gettext** (1), **gettext** (3)