## NAME

Locale::TextDomain − Perl Interface to Uniforum Message Translation

## SYNOPSIS

```
use Locale::TextDomain ('my-package', @locale_dirs);

use Locale::TextDomain qw (my-package);

my $translated = __"Hello World!\n";

my $alt = $__{"Hello World!\n"};

my $alt2 = $__->{"Hello World!\n"};

my @list = (N__"Hello",
            N__"World");

printf (__n ("one file read",
             "%d files read",
             $num_files),
        $num_files);

print __nx ("one file read", "{num} files read", $num_files,
            num => $num_files);

my $translated_context = __p ("Verb, to view", "View");

printf (__np ("Files read from filesystems",
              "one file read",
              "%d files read",
              $num_files),
        $num_files);

print __npx ("Files read from filesystems",
             "one file read",
             "{num} files read",
             $num_files,
             num => $num_files);
```

## DESCRIPTION

The module **Locale::TextDomain** (3pm) provides a high-level interface to Perl message translation.

### Textdomains

When you request a translation for a given string, the system used in libintl-perl follows a standard strategy to find a suitable message catalog containing the translation: Unless you explicitly define a name for the message catalog, libintl-perl will assume that your catalog is called 'messages' (unless you have changed the default value to something else via **Locale::Messages** (3pm), method **textdomain()**).

You might think that his default strategy leaves room for optimization and you are right. It would be a lot smarter if multiple software packages, all with their individual message catalogs, could be installed on one system, and it should also be possible that third-party components of your software (like Perl modules) can load their message catalogs, too, without interfering with yours.

The solution is clear, you have to assign a unique name to your message database, and you have to specify that name at run-time. That unique name is the so-called *textdomain* of your software package. The name is actually arbitrary but you should follow these best-practice guidelines to ensure maximum interoperability:

File System Safety
: In practice, textdomains get mapped into file names, and you should therefore make sure that the textdomain you choose is a valid filename on every system that will run your software.

Case-sensitivity
: Textdomains are always case-sensitive (i. e. 'Package' and 'PACKAGE' are not the same). However, since the message catalogs will be stored on file systems, that may or may not distinguish case when looking up file names, you should avoid potential conflicts here.

Textdomain Should Match CPAN Name
: If your software is listed as a module on CPAN, you should simply choose the name on CPAN as your textdomain. The textdomain for libintl-perl is hence 'libintl−perl'. But please replace all periods ('.') in your package name with an underscore because ...

Internet Domain Names as a Fallback
: ... if your software is *not* a module listed on CPAN, as a last resort you should use the Java(tm) package scheme, i. e. choose an internet domain that you are owner of (or ask the owner of an internet domain) and concatenate your preferred textdomain with the reversed internet domain. Example: Your company runs the web-site 'www.foobar.org' and is the owner of the domain 'foobar.org'. The textdomain for your company's software 'barfoos' should hence be 'org.foobar.barfoos'.

If your software is likely to be installed in different versions on the same system, it is probably a good idea to append some version information to your textdomain.

Other systems are less strict with the naming scheme for textdomains but the phenomena known as Perl is actually a plethora of small, specialized modules and it is probably wisest to postulate some namespace model in order to avoid chaos.

### Binding textdomains to directories

Once the system knows the *textdomain* of the message that you want to get translated into the user's language, it still has to find the correct message catalog. By default, libintl-perl will look up the string in the translation database found in the directories */usr/share/locale* and */usr/local/share/locale* (in that order).

It is neither guaranteed that these directories exist on the target machine, nor can you be sure that the installation routine has write access to these locations. You can therefore instruct libintl-perl to search other directories prior to the default directories. Specifying a different search directory is called *binding* a textdomain to a directory.

Beginning with version 1.20, **Locale::TextDomain** extends the default strategy by a Perl-specific approach. If File::ShareDir is installed, it will look in the subdirectories named *locale* and *LocaleData* (in that order) in the directory returned by `File::ShareDir::dist_dir ($textdomain)` (if File::ShareDir is installed), and check for a database containing the message for your textdomain there. This allows you to install your database in the Perl-specific shared directory using Module::Install's `install_share` directive or the Dist::Zilla ShareDir plugin.

If File::ShareDir is not available, or if Locale::TextDomain fails to find the translation files in the File::ShareDir directory, it will next look in every directory found in the standard include path `@INC`, and check for a database containing the message for your textdomain there. Example: If the path */usr/lib/perl/5.8.0/site_perl* is in your `@INC`, you can install your translation files in */usr/lib/perl/5.8.0/site_perl/LocaleData*, and they will be found at run-time.

## USAGE

It is crucial to remember that you use **Locale::TextDomain** (3) as specified in the section ''SYNOPSIS'', that means you have to **use** it, not **require** it. The module behaves quite differently compared to other modules.

The most significant difference is the meaning of the list passed as an argument to the **use()** function. It actually works like this:

```
use Locale::TextDomain (TEXTDOMAIN, DIRECTORY, ...)
```

The first argument (the first string passed to **use()**) is the textdomain of your package, optionally followed by a list of directories to search *instead* of the Perl-specific directories (see above: */LocaleData* appended to a *File::ShareDir* directory and every path in `@INC`).

If you are the author of a package 'barfoos', you will probably put the line

```
use Locale::TextDomain 'barfoos';
```

resp. for non-CPAN modules

```
use Locale::TextDomain 'org.foobar.barfoos';
```

in every module of your package that contains translatable strings. If your module has been installed properly, including the message catalogs, it will then be able to retrieve these translations at run-time.

If you have not installed the translation database in a directory *LocaleData* in the File::ShareDir directory or the standard include path `@INC` (or in the system directories */usr/share/locale* resp. */usr/local/share/locale*), you have to explicitly specify a search path by giving the names of directories (as strings!) as additional arguments to **use()**:

```
use Locale::TextDomain qw (barfoos ./dir1 ./dir2);
```

Alternatively you can call the function **bindtextdomain()** with suitable arguments (see the entry for **bindtextdomain()** in "FUNCTIONS" in Locale::Messages). If you do so, you should pass `undef` as an additional argument in order to avoid unnecessary lookups:

```
use Locale::TextDomain ('barfoos', undef);
```

You see that the arguments given to **use()** have nothing to do with what is imported into your namespace, but they are rather arguments to **textdomain()**, resp. **bindtextdomain()**. Does that mean that **Locale::TextDomain** exports nothing into your namespace? Umh, not exactly ... in fact it imports *all* functions listed below into your namespace, and hence you should not define conflicting functions (and variables) yourself.

So, why has Locale::TextDomain to be different from other modules? If you have ever written software in C and prepared it for internationalization (i18n), you will probably have defined some preprocessor macros like:

```
#define _(String) dgettext ("my-textdomain", String)
#define N_(String) String
```

You only have to define that once in C, and the textdomain for your package is automatically inserted into all gettext functions. In Perl there is no such mechanism (at least it is not portable, option -P) and using the gettext functions could become quite cumbersome without some extra fiddling:

```
print dgettext ("my-textdomain", "Hello world!\n");
```

This is no fun. In C it would merely be a

```
printf (_("Hello world!\n"));
```

Perl has to be more concise and shorter than C ... see the next section for how you can use **Locale::TextDomain** to end up in Perl with a mere

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

## EXPORTED FUNCTIONS

All functions have quite funny names on purpose. In fact the purpose for that is quite clear: They should be short, operator-like, and they should not yell for conflicts with existing functions in *your* namespace. You will understand it, when you internationalize your first Perl program or module. Preparing it is more like marking strings as being translatable than inserting function calls. Here we go:

### __ MSGID

**NOTE:** This is a *double* underscore!

The basic and most-used function. It is a short-cut for a call to **gettext()** resp. **dgettext()**, and simply returns the translation for **MSGID**. If your old code reads like this:

```
print "permission denied";
```

You will now write:

```
print __"permission denied";
```

That's all, the string will be output in the user's preferred language, provided that you have installed a translation for it.

Of course you can also use parentheses:

```
print __("permission denied");
```

Or even:

```
print (__("permission denied"));
```

In my eyes, the first version without parentheses looks best.

**__x MSGID, ID1 => VAL1, ID2 => VAL2, ...**

One of the nicest features in Perl is its capability to interpolate variables into strings:

```
print "This is the $color $thing.\n";
```

This nice feature might con you into thinking that you could now write

```
print __"This is the $color $thing.\n";
```

Alas, that would be nice, but it is not possible. Remember that the function __() serves both as an operator for translating strings *and* as a mark for translatable strings. If the above string would get extracted from your Perl code, the un-interpolated form would end up in the message catalog because when parsing your code it is unpredictable what values the variables `$thing` and `$color` will have at run-time (this fact is most probably one of the reasons you have written your program for).

However, at run-time, Perl will have interpolated the values already *before* __() (resp. the underlying **gettext()** function) has seen the original string. Consequently something like ''This is the red car.\n'' will be looked up in the message catalog, it will not be found (because only ''This is the `$color $thing`.\n'' is included in the database), and the original, untranslated string will be returned. Honestly, because this is almost always an error, the **xgettext** (1) program will bail out with a fatal error when it comes across that string in your code.

There are two workarounds for that:

```
printf __"This is the %s %s.\n", $color, $thing;
```

But that has several disadvantages: Your translator will only see the isolated string, and without the surrounding code it is almost impossible to interpret it correctly. Of course, GNU emacs and other software capable of editing PO translation files will allow you to examine the context in the source code, but it is more likely that your translator will look for a less challenging translation project when she frequently comes across such messages.

And even if she does understand the underlying programming, what if she has to reorder the color and the thing like in French:

```
msgid "This is the red car.\n";
msgstr "Cela est la voiture rouge.\n"
```

Zut alors! While it is possible to reorder the arguments to **printf()** and friends, it requires a syntax that is is nothing that you want to learn.

So what? The Perl backend to GNU gettext has defined an alternative format for interpolatable strings:

```
"This is the {color} {thing}.\n";
```

Instead of Perl variables you use place-holders (legal Perl variables are also legal place-holders) in curly braces, and then you call

```
print __x ("This is the {color} {thing}.\n",
           thing => $thang,
           color => $color);
```

The function **__x()** will take the additional hash and replace all occurencies of the hash keys in curly braces with the corresponding values. Simple, readable, understandable to translators, what else would you want? And if the translator forgets, misspells or otherwise messes up some ‘‘variables’’, the **msgfmt**(1) program, that is used to compile the textual translation file into its binary representation will even choke on these errors and refuse to compile the translation.

**__n MSGID, MSGID_PLURAL, COUNT**

Whew! That looks complicated ... It is best explained with an example. We’ll have another look at your vintage code:

```
if ($files_deleted > 1) {
    print "All files have been deleted.\n";
} else {
    print "One file has been deleted.\n";
}
```

Your intent is clear, you wanted to avoid the cumbersome ‘‘1 files deleted’’. This is okay for English, but other languages have more than one plural form. For example in Russian it makes a difference whether you want to say 1 file, 3 files or 6 files. You will use three different forms of the noun ’file’ in each case. [Note: Yep, very smart you are, the Russian word for ’file’ is in fact the English word, and it is an invariable noun, but if you know that, you will also understand the rest despite this little simplification ...].

That is the reason for the existence of the function **ngettext()**, that **__n()** is a short-cut for:

```
print __n"One file has been deleted.\n",
         "All files have been deleted.\n",
         $files_deleted;
```

Alternatively:

```
print __n ("One file has been deleted.\n",
           "All files have been deleted.\n",
           $files_deleted);
```

The effect is always the same: libintl-perl will find out which plural form to pick for your user’s language, and the output string will always look okay.

**__nx MSGID, MSGID_PLURAL, COUNT, VAR1 => VAL1, VAR2 => VAL2, ...**

Bringing it all together:

```
print __nx ("One file has been deleted.\n",
            "{count} files have been deleted.\n",
            $num_files,
            count => $num_files);
```

The function **__nx()** picks the correct plural form (also for English!) *and* it is capable of interpolating variables into strings.

Have a close look at the order of arguments: The first argument is the string in the singular, the second one is the plural string. The third one is an integer indicating the number of items. This third argument is *only* used to pick the correct translation. The optionally following arguments make up the hash used for interpolation. In the beginning it is often a little confusing that the variable holding the number of items will usually be repeated somewhere in the interpolation hash.

**__xn MSGID, MSGID_PLURAL, COUNT, VAR1 => VAL1, VAR2 => VAL2, ...**

Does exactly the same thing as **__nx()**. In fact it is a common typo promoted to a feature.

**__p MSGCTXT, MSGID**

This is much like __. The ‘‘p’’ stands for ‘‘particular’’, and the MSGCTXT is used to provide context to the translator. This may be necessary when your string is short, and could stand for multiple things. For example:

```
print __p"Verb, to view", "View";
print __p"Noun, a view", "View";
```

The above may be ‘‘View’’ entries in a menu, where View−>Source and File−>View are different forms of ‘‘View’’, and likely need to be translated differently.

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:

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

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

This function was introduced in libintl-perl 1.17.

**__px MSGCTXT, MSGID, VAR1 => VAL1, VAR2 => VAL2, ...**

Like **__p()**, b ut supports variable substitution in the string, like **__x()**.

```
print __px("Verb, to view", "View {file}", file => $filename);
```

See **__p()** and **__x()** for more details.

This function was introduced in libintl-perl 1.17.

**__np MSGCTXT, MSGID, MSGID_PLURAL, COUNT**

This adds context to plural calls. It should not be needed very often, if at all, due to the **__nx()** function. The type of variable substitution used in other gettext libraries (using sprintf-like sybols, like `%s` or `%1`) sometimes required context. For a (bad) example of this:

```
printf (__np("[count] files have been deleted",
             "One file has been deleted.\n",
             "%s files have been deleted.\n",
             $num_files),
        $num_files);
```

NOTE: The above usage is discouraged. Just use the **__nx()** call, which pro vides inline context via the key names.

This function was introduced in libintl-perl 1.17.

**__npx MSGCTXT, MSGID, MSGID_PLURAL, COUNT, VAR1 => VAL1, VAR2 => VAL2, ...**

This is provided for comleteness. It adds the variable interpolation into the string to the previous method, **__np()**.

It’s usage would be like so:

```
print __npx ("Files being permenantly removed",
             "One file has been deleted.\n",
             "{count} files have been deleted.\n",
             $num_files,
             count => $num_files);
```

I cannot think of any situations requiring this, but we can easily support it, so here it is.

This function was introduced in libintl-perl 1.17.

**N_ _ (ARG1, ARG2, ...)**
A no-op function that simply echoes its arguments to the caller. Take the following piece of Perl:

```
my @options = (
    "Open",
    "Save",
    "Save As",
);

...

my $option = $options[1];
```

Now say that you want to have this translatable. You could sometimes simply do:

```
my @options = (
    __"Open",
    __"Save",
    __"Save As",
);

...

my $option = $options[1];
```

But often times this will not be what you want, for example when you also need the unmodified original string. Sometimes it may not even work, for example, when the preferred user language is not yet determined at the time that the list is initialized.

In these cases you would write:

```
my @options = (
    N__"Open",
    N__"Save",
    N__"Save As",
);

...

my $option = __($options[1]);
# or: my $option = dgettext ('my-domain', $options[1]);
```

Now all the strings in `@options` will be left alone, since N_ _() returns its arguments (one ore more) unmodified. Nevertheless, the string extractor will be able to recognize the strings as being translatable. And you can still get the translation later by passing the variable instead of the string to one of the above translation functions.

**N_ _n (MSGID, MSGID_PLURAL, COUNT)**
Does exactly the same as N__(). You will use this form if you have to mark the strings as having plural forms.

**N_ _p (MSGCTXT, MSGID)**
Marks **MSGID** as N__() does, but in the context **MSGCTXT**.

**N_ _np (MSGCTXT, MSGID, MSGID_PLURAL, COUNT)**
Marks **MSGID** as N**__n()** does, but in the context **MSGCTXT**.

## EXPORTED VARIABLES

The module exports several variables into your namespace:

**%__**

A tied hash. Its keys are your original messages, the values are their translations:

```
my $title = "<h1>$__{'My Homepage'}</h1>";
```

This is much better for your translation team than

```
my $title = __"<h1>My Homepage</h1>";
```

In the second case the HTML code will make it into the translation database and your translators have to be aware of HTML syntax when translating strings.

**Warning:** Do *not* use this hash outside of double-quoted strings! The code in the tied hash object relies on the correct working of the function **caller()** (see ‘‘perldoc −f caller’’), and this function will report incorrect results if the tied hash value is the argument to a function from another package, for example:

```
my $result = Other::Package::do_it ($__{'Some string'});
```

The tied hash code will see ‘‘Other::Package’’ as the calling package, instead of your own package. Consequently it will look up the message in the wrong text domain. There is no workaround for this bug. Therefore:

Never use the tied hash interpolated strings!

**$__**

A reference to %__, in case you prefer:

```
my $title = "<h1>$__->{'My Homepage'}</h1>";
```

## PERFORMANCE

Message translation can be a time-consuming task. Take this little example:

```
1: use Locale::TextDomain ('my−domain');
2: use POSIX (:locale_h);
3:
4: setlocale (LC_ALL, '');
5: print __"Hello world!\n";
```

This will usually be quite fast, but in pathological cases it may run for several seconds. A worst-case scenario would be a Chinese user at a terminal that understands the codeset Big5−HKSCS. Your translator for Chinese has however chosen to encode the translations in the codeset EUC-TW.

What will happen at run-time? First, the library will search and load a (maybe large) message catalog for your textdomain ’my−domain’. Then it will look up the translation for ‘‘Hello world!\n’’, it will find that it is encoded in EUC-TW. Since that differs from the output codeset Big5−HKSCS, it will first load a conversion table containing several ten-thousands of codepoints for EUC-TW, then it does the same with the smaller, but still very large conversion table for Big5−HKSCS, it will convert the translation on the fly from EUC-TW into Big5−HKSCS, and finally it will return the converted translation.

A worst-case scenario but realistic. And for these five lines of codes, there is not much you can do to make it any faster. You should understand, however, *when* the different steps will take place, so that you can arrange your code for it.

You have learned in the section ‘‘DESCRIPTION’’ that line 1 is responsible for locating your message database. However, the **use()** will do nothing more than remembering your settings. It will not search any directories, it will not load any catalogs or conversion tables.

Somewhere in your code you will always have a call to **POSIX::setlocale()**, and the performance of this call may be time-consuming, depending on the architecture of your system. On some systems, this will consume very little time, on others it will only consume a considerable amount of time for the first call, and on others it may always be time-consuming. Since you cannot know, how **setlocale()** is implemented on the target system, you should reduce the calls to **setlocale()** to a minimum.

Line 5 requests the translation for your string. Only now, the library will actually load the message catalog, and only now will it load eventually needed conversion tables. And from now on, all this information will be cached in memory. This strategy is used throughout libintl-perl, and you may describe it as 'load−on−first−access'. Getting the next translation will consume very little resources.

However, although the translation retrieval is somewhat obfuscated by an operator-like function call, it is still a function call, and in fact it even involves a chain of function calls. Consequently, the following example is probably bad practice:

```
foreach (1 .. 100_000) {
    print __"Hello world!\n";
}
```

This example introduces a lot of overhead into your program. Better do this:

```
my $string = __"Hello world!\n";
foreach (1 .. 100_000) {
    print $string;
}
```

The translation will never change, there is no need to retrieve it over and over again. Although libintl-perl will of course cache the translation read from the file system, you can still avoid the overhead for the function calls.

## AUTHOR



## SEE ALSO

**Locale::Messages** (3pm), **Locale::gettext_pp** (3pm), **perl** (1), **gettext** (1), **gettext** (3)