NAME

Locale::TextDomain::FAQ − Frequently asked questions for libintl−perl

DESCRIPTION

This FAQ

QUESTIONS AND ANSWERS

Why is libintl-perl so big? Why don't you use Encode (3pm) for character set conversion instead of rolling your own version?

Encode (3pm) requires at least Perl 5.7.x, whereas libintl-perl needs to be operational on Perl 5.004. Internally, libintl-perl uses **Encode** (3pm) if it is available.

Why do the gettext functions always unset the utf−8 flag on the strings it returns?

Because the gettext functions do not know whether the string is encoded in utf−8 or not. Instead of taking guesses, it rather unsets the flag.

Can I set the utf−8 flag on strings returned by the gettext family of functions?

Yes, but it is not recommended. If you absolutely want to do it, use the function bind_textdomain_filter in Locale::Messages for it.

The strings returned by gettext and friends are by default encoded in the preferred charset for the user's locale, but there is no portable way to find out, whether this is utf−8 or not. That means, you either have to enforce utf−8 as the output character set (by means of **bind_textdomain_codeset()** and/or the environment variable OUTPUT_CHARSET) and override the user preference, or you run the risk of marking strings as utf−8 which really aren't utf−8.

The whole concept behind that utf−8 flag introduced in Perl 5.6 is seriously broken, and the above described dilemma is a proof for that. The best thing you can do with that flag is get rid of it, and turn it off. Your code will benefit from it and become less error prone, more portable and faster.

Why do non-ASCII characters in my Gtk2 application look messed up?

The Perl binding of Gtk2 has a design flaw. It expects all UI messages to be in UTF−8 and it also expects messages to be flagged as utf−8. The only solution for you is to enforce all your po files to be encoded in utf−8 (convert them manually, if you need to), and also enforce that charset in your application, regardless of the user's locale settings. Assumed that your textdomain is ''org.bar.foo'', you have to code the following into your main module or script:

```
BEGIN {
    bind_textdomain_filter 'org.bar.foo', \&turn_utf_8_on;
    bind_textdomain_codeset 'org.bar.foo', 'utf−8';
}
```

See the File GTestRunner.pm of **Test::Unit::GTestRunner** (3pm) for details.

How do I interface Glade2 UI definitions with libintl-perl?

Gtk2::GladeXML (3pm) seems to ignore calls to **bind_textdomain()**. See the File GTestRunner.pm of **Test::Unit::GTestRunner** (3pm) for a possible solution.

Why does Locale::TextDomain use a double underscore? I am used to a single underscore from C or other languages.

Function names that consist of exactly one non-alphanumerical character make the function automatically global in Perl. Besides, in Perl 6 the concatenation operator will be the underscore instead of the dot.

How do I switch languages or force a certain language independently from user settings read from the environment?

The simple answer is:

```
use POSIX qw (setlocale LC_ALL);

my $language = 'fr';
my $country = 'FR';
my $charset = 'iso−8859−1';
```

```
    setlocale LC_ALL, "${language}_$country.$charset";
```

Sadly enough, this will fail in many cases. The problem is that locale identifiers are not standardized and are completely system-dependent. Not only their overall format, but also other details like case-sensitivity. Some systems are very forgiving about the system − for example normalizing charset descriptions − others very strict. In order to be reasonably platform independent, you should try a list of possible locale identifiers for your desired settings. This is about what I would try for achieving the above:

```
   my @tries = qw (
        fr_FR.iso−8859−1 fr_FR.iso8859−1 fr_FR.iso88591
        fr_FR.ISO−8859−1 fr_FR.ISO8859−1 fr_FR.ISO88591
        fr.iso−8859−1 fr.iso8859−1 fr.iso88591
        fr.ISO−8859−1 fr.ISO8859−1 fr.ISO88591
        fr_FR
        French_France.iso−8859−1 French_France.iso8859−1 French_France.iso88591
        French_France.ISO−8859−1 French_France.ISO8859−1 French_France.ISO88591
        French.iso−8859−1 French.iso8859−1 French.iso88591
        French.ISO−8859−1 French.ISO8859−1 French.ISO88591
   );
   foreach my $try (@tries) {
        last if setlocale LC_ALL, $try;
   }
```

Set **Locale::Util** (3pm) for functions that help you with this.

Alternatively, you can force a certain language by setting the environment variables LANGUAGE, LANG and OUTPUT_CHARSET, but this is only guaranteed to work, if you use the pure Perl implementation of gettext (see the documentation for **select_package()** in **Locale::Messages** (3pm)). You would do the above like this:

```
    use Locale::Messages qw (nl_putenv);

    # LANGUAGE is a colon separated list of languages.
    nl_putenv("LANGUAGE=fr_FR");

    # If LANGUAGE is set, LANG should be set to the primary language.
    # This is not needed for gettext, but for other parts of the system
    # it is.
    nl_putenv("LANG=fr_FR");

    # Force an output charset like this:
    nl_putenv("OUTPUT_CHARSET=iso−8859−1");

    setlocale (LC_MESSAGES, 'C');
```

These environment variables are GNU extensions, and they are also honored by libintl-perl. Still, you should always try to set the locale with setlocale for the catch-all category LC_ALL. If you miss to do so, your program's output maybe cluttered, mixing languages and charsets, if the system runs in a locale that is not compatible with your own language settings.

Remember that these environment variables are not guaranteed to work, if you use an XS version of gettext. In order to force usage of the pure Perl implementation, do the following:

```
    Locale::Messages−>select_package ('gettext_pp');
```

If you think, this is brain-damaged, you are right, but I cannot help you. Actually there should be a more flexible API than setlocale, but at the time of this writing there isn't. Until then, the recommentation goes like this:

```
1) Try setting LC_ALL with Locale::Util.
2) If that does not succeed, either give up or ...
3) Reset LC_MESSAGES to C/POSIX.
4) Switch to pure Perl for gettext.
5) Set the environment variables LANGUAGE, LANG,
   and OUTPUT_CHARSET to your desired values.
```

What is the advantage of libintl-perl over Locale::Maketext?

Of course, I can only give my personal opinion as an answer.

Locale::Maketext claims to fix design flaws in gettext. These alleged design flaws, however, boil down to one pathological case which always has a workaround. But both programmers and translators pay this fix with an unnecessarily complicated interface.

The paramount advantage of libintl-perl is that it uses an approved technology and concept. Except for Java(tm) programs, this is the state-of-the-art concept for localizing Un*x software. Programmers that have already localized software in C, C++, C#, Python, PHP, or a number of other languages will feel instantly at home, when localizing software written in Perl with libintl-perl. The same holds true for the translators, because the files they deal with have exactly the same format as those for other programming languages. They can use the same set of tools, and even the commands they have to execute are the same.

With libintl-perl refactoring of the software is painless, even if you modify, add or delete translatable strings. The gettext tools are powerful enough to reduce the effort of the translators to the bare minimum. Maintaining the message catalogs of Locale::Maketext in larger scale projects, is IMHO unfeasible.

Editing the message catalogs of Locale::Maketext − they are really Perl modules − asks too much from most translators, unless they are programmers. The portable object (po) files used by libintl-perl have a simple syntax, and there are a bunch of specialized GUI editors for these files, that facilitate the translation process and hide most complexity from the user.

Furthermore, libintl-perl makes it possible to mix programming languages without a paradigm shift in localization. Without any special efforts, you can write a localized software that has modules written in C, modules in Perl, and builds a Gtk user interface with Glade. All translatable strings end up in one single message catalog.

Last but not least, the interface used by libintl-perl is plain simple: Prepend translatable strings with a double underscore, and you are done in most cases.

Why do single-quoted strings not work?

You probably write something like this:

```
print __'Hello';
```

And you get an error message like ‘‘Can’t find string terminator ’’’‘ anywhere before EOF at ...’’, or even ‘‘Bareword found where operator expected at ... Might be a runaway multi-line ’’ string starting on’’. The above line is (really!) essentially the same as writing:

```
print __::Hello';
```

A lesser know feature of Perl is that you can use a single quote (‘‘’’) as the separator in packages instead of the double colon (‘‘:’’). What the Perl parser sees in the first example is a valid package name (‘‘__’’) followed by the separator (‘‘’’), then another valid package name (‘‘Hello’’) followed by a lone single quote. It is therefore not a problem in libintl-perl but simple wrong Perl syntax. You have to correct alternatives:

```
print __ 'Hello';   # Insert a space to disambiguate.
```

Or use double-quotes:

```
print __"Hello";
```

Thanks to Slavi Agafonkin for pointing me to the solution of this mystery.