NAME

Locale::gettext_dumb − Locale unaware Implementation of Uniforum Message Translation

SYNOPSIS

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

# Normally, you will not want to include this module directly but this way:
use Locale::Messages;

my $selected = Locale::Messages−>select_package ('gettext_dumb');

gettext $msgid;
dgettext $domainname, $msgid;
dcgettext $domainname, $msgid, LC_MESSAGES;
ngettext $msgid, $msgid_plural, $count;
dngettext $domainname, $msgid, $msgid_plural, $count;
dcngettext $domainname, $msgid, $msgid_plural, $count, LC_MESSAGES;
pgettext $msgctxt, $msgid;
dpgettext $domainname, $msgctxt, $msgid;
dcpgettext $domainname, $msgctxt, $msgid, LC_MESSAGES;
npgettext $msgctxt, $msgid, $msgid_plural, $count;
dnpgettext $domainname, $msgctxt, $msgid, $msgid_plural, $count;
dcnpgettext $domainname, $msgctxt, $msgid, $msgid_plural, $count, LC_MESSAGES;
textdomain $domainname;
bindtextdomain $domainname, $directory;
bind_textdomain_codeset $domainname, $encoding;
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

IMPORTANT! This module is experimental. It may not work as described!

The module **Locale::gettext_dumb** does exactly the same as **Locale::gettext_xs** (3pm) or **Locale::gettext_pp** (3pm).

While both other modules use **POSIX::setlocale()** to determine the currently selected locale, this backend only checks the environment variables LANGUAGE, LANG, LC_ALL, LC_MESSAGES (in that order), when it tries to locate a message catalog (a .mo file).

This class was introduced in libintl-perl 1.22.

USAGE

This module should not be used for desktop software or scripts run locally. Why? If you use a message catalog for example in Danish in UTF−8 (da_DA.UTF8) but the system locale is set to Russian with KOI8−R (ru_RU.KOI8−R) you may produce invalid output, either invalid multi-byte sequences or invalid text, depending on how you look at it.

That will happen, when you mix output from **Locale::gettext_pp** with locale-dependent output from the operating system like the contents of the variable “$!”, date and time formatting functions (**localtime()**, **gmtime()**, **POSIX::strftime()** etc.), number formatting with **printf()** and friends, and so on.

A typical usage scenario looks like this:

You have a server application (for example a web application) that is supposed to display a fixed set of messages in many languages. If you want to do this with **Locale::gettext_xs** (3pm) or

Locale::gettext_pp (3pm), you have to install the locale data for all of those languages. Otherwise, translating the messages will not work.

With **Locale::gettext_dumb** (3pm) you can relax these requirements, and display messages for all languages that you have mo files for.

On the other hand, you will soon reach limits with this approach. Almost any application requires more than bare translation of messages for localisation. You want to formatted dates and times, you want to display numbers in the correct formatting for the selected languages, and you may want to display system error messages (“$!”).

In practice, **Locale::gettext_dumb** (3pm) is still useful in these scenarios. Your users will have to live with the fact that the presented output is in different languages resp. for different locales, when ‘‘their’’ locale is not installed on your system.

More dangerous is mixing output in different character sets but that can be easily avoided. Simply make sure that **Locale::gettext_dump** uses UTF−8 (for example by setting the environment variable OUTPUT_CHARSET or by calling **bind_textdomain_codeset()**) and make sure that the system locale also uses UTF−8, for example ‘‘en_US.UTF8’’. If that fails, switch to a locale that uses a subset of UTF−8. In practice that will be US-ASCII, the character set used by the default locale ‘‘C’’ resp. ‘‘POSIX’’.

Your application will then to a certain extent mix output for different localisations resp. languages. But this is completely under your control.

EXAMPLE

See above! Normally you should not use this module! However, let us assume you have read the warnings. In a web application you would do something like this:

```
use Locale::TextDomain qw (com.example.yourapp);
use Locale::Messages qw (nl_putenv LC_ALL bindtextdomain
                         bind_textdomain_codeset);
use Locale::Util qw (web_set_locale);
use POSIX qw (setlocale);

# First try to switch to the locale requested by the user.  If you
# know it you can try to pass it to setlocale like this:
#
#   my $hardcoded_locale = 'fr_FR.UTF−8';
#   my $success = POSIX::setlocale (LC_ALL, $hardcoded_locale);
#
# However, we try to let libintl−perl do a better job for us:
my $success = web_set_locale $ENV{HTTP_ACCEPT_LANGUAGE},
                             $ENV{HTTP_ACCEPT_CHARSET};
# Note: If your application forces the use of UTF−8 for its output
# you should pass 'UTF−8' as the second argument to web_set_locale
# instead of $ENV{HTTP_ACCEPT_CHARSET}.

if (!$success) {
    # Did not work.  Switch to the dumb interface of
    # Locale::Messages.
    Locale::Messages−>select_package ('gettext_dumb');

    # And try to switch to a default locale:
    if (!setlocale (LC_ALL, 'en_US.UTF−8')) {
        # Still no luck.  Enforce at least US−ASCII:
        setlocale (LC_ALL, 'C');
    }
    bind_textdomain_codeset 'com.example.yourapp', 'utf−8';
}
```

If your application forces the usage of UTF−8 you should ignore the environment variable

AUTHOR

Copyright (C) 2002−2016 Guido Flohr <http://www.guido-flohr.net/> (<mailto:guido.flohr@cantanea.com>), all rights reserved. See the source code for details!code for details!

SEE ALSO

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