NAME

encoding - allows you to write your script in non-ASCII and non-UTF-8

WARNING

This module has been deprecated since perl v5.18. See *DESCRIPTION* and *BUGS*.

SYNOPSIS

```
use encoding "greek";  # Perl like Greek to you?
use encoding "euc-jp"; # Jperl!

# or you can even do this if your shell supports your native encoding

perl -Mencoding=latin2 -e'...' # Feeling centrally European?
perl -Mencoding=euc-kr -e'...' # Or Korean?

# more control

# A simple euc-cn => utf-8 converter
use encoding "euc-cn", STDOUT => "utf8";  while(<>){print};

# "no encoding;" supported
no encoding;

# an alternate way, Filter
use encoding "euc-jp", Filter=>1;
# now you can use kanji identifiers -- in euc-jp!

# encode based on the current locale - specialized purposes only;
# fraught with danger!!
use encoding ':locale';
```

DESCRIPTION

This pragma is used to enable a Perl script to be written in encodings that aren't strictly ASCII nor UTF-8. It translates all or portions of the Perl program script from a given encoding into UTF-8, and changes the PerlIO layers of `STDIN` and `STDOUT` to the encoding specified.

This pragma dates from the days when UTF-8-enabled editors were uncommon. But that was long ago, and the need for it is greatly diminished. That, coupled with the fact that it doesn't work with threads, along with other problems, (see *BUGS*) have led to its being deprecated. It is planned to remove this pragma in a future Perl version. New code should be written in UTF-8, and the `use utf8` pragma used instead (see *perluniintro* and *utf8* for details). Old code should be converted to UTF-8, via something like the recipe in the *SYNOPSIS* (though this simple approach may require manual adjustments afterwards).

The only legitimate use of this pragma is almost certainly just one per file, near the top, with file scope, as the file is likely going to only be written in one encoding. Further restrictions apply in Perls before v5.22 (see *Prior to Perl v5.22*).

There are two basic modes of operation (plus turning if off):

`use encoding ['`*ENCNAME*`'] ;`

> This is the normal operation. It translates various literals encountered in the Perl source file from the encoding *ENCNAME* into UTF-8, and similarly converts character code points. This is used when the script is a combination of ASCII (for the variable names and punctuation, *etc*),

but the literal data is in the specified encoding.

ENCNAME is optional. If omitted, the encoding specified in the environment variable *PERL_ENCODING* is used. If this isn't set, or the resolved-to encoding is not known to *Encode*, the error `Unknown encoding 'ENCNAME'` will be thrown.

Starting in Perl v5.8.6 (`Encode` version 2.0.1), *ENCNAME* may be the name `:locale`. This is for very specialized applications, and is documented in *The `:locale` sub-pragma* below.

The literals that are converted are `q//, qq//, qr//, qw///, qx//`, and starting in v5.8.1, `tr///`. Operations that do conversions include `chr`, `ord`, `utf8::upgrade` (but not `utf8::downgrade`), and `chomp`.

Also starting in v5.8.1, the `DATA` pseudo-filehandle is translated from the encoding into UTF-8.

For example, you can write code in EUC-JP as follows:

```
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
             #<-char-><-char->   # 4 octets
s/\bCamel\b/$Rakuda/;
```

And with `use encoding "euc-jp"` in effect, it is the same thing as that code in UTF-8:

```
my $Rakuda = "\x{99F1}\x{99DD}"; # two Unicode Characters
s/\bCamel\b/$Rakuda/;
```

See *EXAMPLE* below for a more complete example.

Unless `${^UNICODE}` (available starting in v5.8.2) exists and is non-zero, the PerlIO layers of `STDIN` and `STDOUT` are set to "`:encoding(`*ENCNAME*`)`". Therefore,

```
use encoding "euc-jp";
my $message = "Camel is the symbol of perl.\n";
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
$message =~ s/\bCamel\b/$Rakuda/;
print $message;
```

will print

```
 "\xF1\xD1\xF1\xCC is the symbol of perl.\n"
```

not

```
 "\x{99F1}\x{99DD} is the symbol of perl.\n"
```

You can override this by giving extra arguments; see below.

Note that `STDERR` WILL NOT be changed, regardless.

Also note that non-STD file handles remain unaffected. Use `use open` or `binmode` to change the layers of those.

`use encoding` *ENCNAME* `Filter=>1;`

This operates as above, but the `Filter` argument with a non-zero value causes the entire script, and not just literals, to be translated from the encoding into UTF-8. This allows identifiers in the source to be in that encoding as well. (Problems may occur if the encoding is not a superset of ASCII; imagine all your semi-colons being translated into something different.) One can use this form to make

```
 ${"\x{4eba}"}++
```

work. (This is equivalent to $*human*++, where *human* is a single Han ideograph).

This effectively means that your source code behaves as if it were written in UTF-8 with `'use utf8'` in effect. So even if your editor only supports Shift_JIS, for example, you can still try examples in Chapter 15 of `Programming Perl, 3rd Ed.`.

This option is significantly slower than the other one.

`no encoding;`

Unsets the script encoding. The layers of `STDIN`, `STDOUT` are reset to "`:raw`" (the default unprocessed raw stream of bytes).

OPTIONS

Setting STDIN and/or STDOUT individually

The encodings of `STDIN` and `STDOUT` are individually settable by parameters to the pragma:

```
use encoding 'euc-tw', STDIN => 'greek'  ...;
```

In this case, you cannot omit the first *ENCNAME*. `STDIN => undef` turns the I/O transcoding completely off for that filehandle.

When `${^UNICODE}` (available starting in v5.8.2) exists and is non-zero, these options will be completely ignored. See *"${^UNICODE}" in perlvar* and *"-C" in perlrun* for details.

The :locale sub-pragma

Starting in v5.8.6, the encoding name may be `:locale`. This means that the encoding is taken from the current locale, and not hard-coded by the pragma. Since a script really can only be encoded in exactly one encoding, this option is dangerous. It makes sense only if the script itself is written in ASCII, and all the possible locales that will be in use when the script is executed are supersets of ASCII. That means that the script itself doesn't get changed, but the I/O handles have the specified encoding added, and the operations like `chr` and `ord` use that encoding.

The logic of finding which locale `:locale` uses is as follows:

1. If the platform supports the `langinfo(CODESET)` interface, the codeset returned is used as the default encoding for the open pragma.

2. If 1. didn't work but we are under the locale pragma, the environment variables `LC_ALL` and `LANG` (in that order) are matched for encodings (the part after "`.`", if any), and if any found, that is used as the default encoding for the open pragma.

3. If 1. and 2. didn't work, the environment variables `LC_ALL` and `LANG` (in that order) are matched for anything looking like UTF-8, and if any found, `:utf8` is used as the default encoding for the open pragma.

If your locale environment variables (`LC_ALL`, `LC_CTYPE`, `LANG`) contain the strings 'UTF-8' or 'UTF8' (case-insensitive matching), the default encoding of your `STDIN`, `STDOUT`, and `STDERR`, and of **any subsequent file open**, is UTF-8.

CAVEATS

SIDE EFFECTS

- If the `encoding` pragma is in scope then the lengths returned are calculated from the length of `$/` in Unicode characters, which is not always the same as the length of `$/` in the native encoding.

- Without this pragma, if strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will be created by decoding the byte strings as *ISO 8859-1 (Latin-1)*.

  The **encoding** pragma changes this to use the specified encoding instead. For example:

  ```
  use encoding 'utf8';
  my $string = chr(20000); # a Unicode string
  utf8::encode($string);   # now it's a UTF-8 encoded byte string
  # concatenate with another Unicode string
  ```

```
print length($string . chr(20000));
```

Will print 2, because `$string` is upgraded as UTF-8. Without `use encoding 'utf8';`, it will print 4 instead, since `$string` is three octets when interpreted as Latin-1.

DO NOT MIX MULTIPLE ENCODINGS

Notice that only literals (string or regular expression) having only legacy code points are affected: if you mix data like this

```
\x{100}\xDF
\xDF\x{100}
```

the data is assumed to be in (Latin 1 and) Unicode, not in your native encoding. In other words, this will match in "greek":

```
"\xDF" =~ /\x{3af}/
```

but this will not

```
"\xDF\x{100}" =~ /\x{3af}\x{100}/
```

since the `\xDF` (ISO 8859-7 GREEK SMALL LETTER IOTA WITH TONOS) on the left will **not** be upgraded to `\x{3af}` (Unicode GREEK SMALL LETTER IOTA WITH TONOS) because of the `\x{100}` on the left. You should not be mixing your legacy data and Unicode in the same string.

This pragma also affects encoding of the 0x80..0xFF code point range: normally characters in that range are left as eight-bit bytes (unless they are combined with characters with code points 0x100 or larger, in which case all characters need to become UTF-8 encoded), but if the `encoding` pragma is present, even the 0x80..0xFF range always gets UTF-8 encoded.

After all, the best thing about this pragma is that you don't have to resort to \x{....} just to spell your name in a native encoding. So feel free to put your strings in your encoding in quotes and regexes.

Prior to Perl v5.22

The pragma was a per script, not a per block lexical. Only the last `use encoding` or `no encoding` mattered, and it affected **the whole script**. However, the `no encoding` pragma was supported and `use encoding` could appear as many times as you want in a given script (though only the last was effective).

Since the scope wasn't lexical, other modules' use of `chr`, `ord`, *etc.* were affected. This leads to spooky, incorrect action at a distance that is hard to debug.

This means you would have to be very careful of the load order:

```
# called module
package Module_IN_BAR;
use encoding "bar";
# stuff in "bar" encoding here
1;

# caller script
use encoding "foo"
use Module_IN_BAR;
# surprise! use encoding "bar" is in effect.
```

The best way to avoid this oddity is to use this pragma RIGHT AFTER other modules are loaded. i.e.

```
use Module_IN_BAR;
use encoding "foo";
```

Prior to Encode version 1.87

- STDIN and STDOUT were not set under the filter option. And STDIN=>*ENCODING* and STDOUT=>*ENCODING* didn't work like non-filter version.
- use utf8 wasn't implicitly declared so you have to use utf8 to do

```
${"\x{4eba}"}++
```

Prior to Perl v5.8.1

"NON-EUC" doublebyte encodings

Because perl needs to parse the script before applying this pragma, such encodings as Shift_JIS and Big-5 that may contain '\' (BACKSLASH; \x5c) in the second byte fail because the second byte may accidentally escape the quoting character that follows.

tr///

The **encoding** pragma works by decoding string literals in q//, qq//, qr//, qw///, qx// and so forth. In perl v5.8.0, this does not apply to tr///. Therefore,

```
use encoding 'euc-jp';
#....
$kana =~ tr/\xA4\xA1-\xA4\xF3/\xA5\xA1-\xA5\xF3/;
#           -------- -------- -------- --------
```

Does not work as

```
$kana =~ tr/\x{3041}-\x{3093}/\x{30a1}-\x{30f3}/;
```

Legend of characters above

```
utf8     euc-jp   charnames::viacode()
----------------------------------------
\x{3041} \xA4\xA1 HIRAGANA LETTER SMALL A
\x{3093} \xA4\xF3 HIRAGANA LETTER N
\x{30a1} \xA5\xA1 KATAKANA LETTER SMALL A
\x{30f3} \xA5\xF3 KATAKANA LETTER N
```

This counterintuitive behavior has been fixed in perl v5.8.1.

In perl v5.8.0, you can work around this as follows;

```
use encoding 'euc-jp';
#  ....
eval qq{ \$kana =~ tr/\xA4\xA1-\xA4\xF3/\xA5\xA1-\xA5\xF3/ };
```

Note the tr// expression is surrounded by qq{}. The idea behind this is the same as the classic idiom that makes tr/// 'interpolate':

```
tr/$from/$to/;            # wrong!
eval qq{ tr/$from/$to/ }; # workaround.
```

EXAMPLE - Greekperl

```
use encoding "iso 8859-7";

# \xDF in ISO 8859-7 (Greek) is \x{3af} in Unicode.
```

```
    $a = "\xDF";
    $b = "\x{100}";

    printf "%#x\n", ord($a); # will print 0x3af, not 0xdf

    $c = $a . $b;

    # $c will be "\x{3af}\x{100}", not "\x{df}\x{100}".

    # chr() is affected, and ...

    print "mega\n"  if ord(chr(0xdf)) == 0x3af;

    # ... ord() is affected by the encoding pragma ...

    print "tera\n" if ord(pack("C", 0xdf)) == 0x3af;

    # ... as are eq and cmp ...

    print "peta\n" if "\x{3af}" eq  pack("C", 0xdf);
    print "exa\n"  if "\x{3af}" cmp pack("C", 0xdf) == 0;

    # ... but pack/unpack C are not affected, in case you still
    # want to go back to your native encoding

    print "zetta\n" if unpack("C", (pack("C", 0xdf))) == 0xdf;
```

BUGS

Thread safety

`use encoding ...` is not thread-safe (i.e., do not use in threaded applications).

Can't be used by more than one module in a single program.

Only one encoding is allowed. If you combine modules in a program that have different encodings, only one will be actually used.

Other modules using `STDIN` and `STDOUT` get the encoded stream

They may be expecting something completely different.

literals in regex that are longer than 127 bytes

For native multibyte encodings (either fixed or variable length), the current implementation of the regular expressions may introduce recoding errors for regular expression literals longer than 127 bytes.

EBCDIC

The encoding pragma is not supported on EBCDIC platforms.

`format`

This pragma doesn't work well with `format` because PerlIO does not get along very well with it. When `format` contains non-ASCII characters it prints funny or gets "wide character warnings". To understand it, try the code below.

```
  # Save this one in utf8
```

```
# replace *non-ascii* with a non-ascii string
my $camel;
format STDOUT =
*non-ascii*@>>>>>>>
$camel
.
$camel = "*non-ascii*";
binmode(STDOUT=>':encoding(utf8)'); # bang!
write;              # funny
print $camel, "\n"; # fine
```

Without binmode this happens to work but without binmode, print() fails instead of write().

At any rate, the very use of `format` is questionable when it comes to unicode characters since you have to consider such things as character width (i.e. double-width for ideographs) and directions (i.e. BIDI for Arabic and Hebrew).

See also *CAVEATS*

HISTORY

This pragma first appeared in Perl v5.8.0. It has been enhanced in later releases as specified above.

SEE ALSO

perlunicode, *Encode*, *open*, *Filter::Util::Call*,

Ch. 15 of `Programming Perl (3rd Edition)` by Larry Wall, Tom Christiansen, Jon Orwant; O'Reilly & Associates; ISBN 0-596-00027-8