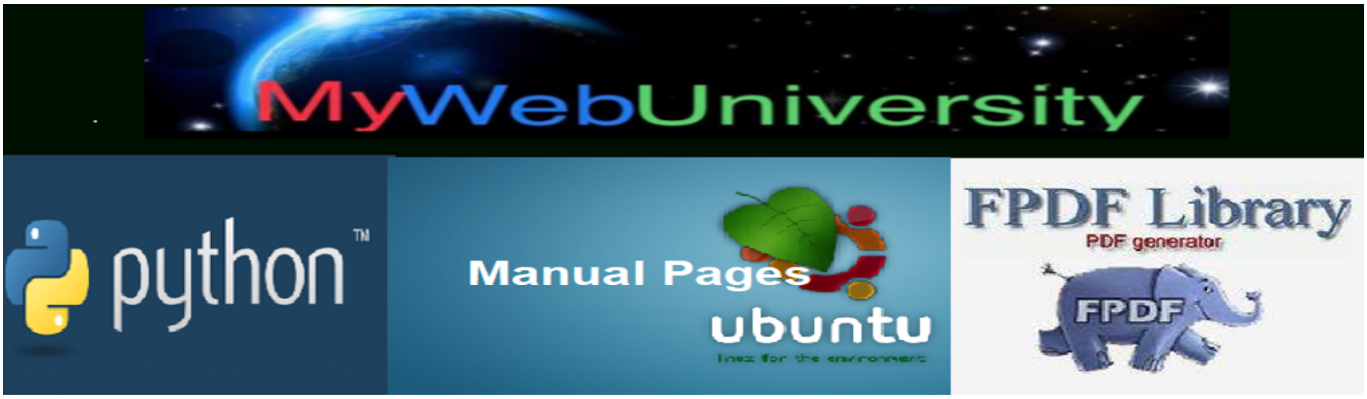

Rocky Enterprise Linux 9.2 Manual Pages on command 'Encode::PerlIO.3perl'

$ man Encode::PerlIO.3perl

NAME

Encode::PerlIO -- a detailed document on Encode and PerlIO

Overview

It is very common to want to do encoding transformations when reading or writing files, network connections, pipes etc. If Perl is configured to use the new 'perlio' IO system then "Encode" provides a "layer" (see PerlIO) which can transform data as it is read or written.

Here is how the blind poet would modernise the encoding:

```
use Encode;
open(my $iliad,'<:encoding(iso-8859-7)','iliad.greek');
open(my $utf8,'>:utf8','iliad.utf8');
my @epic = <$iliad>;
print $utf8 @epic;
close($utf8);
close($illiad);
```

In addition, the new IO system can also be configured to read/write UTF-8 encoded characters (as noted above, this is efficient):

```
open(my $fh,'>:utf8','anything');
print $fh "Any \x{0021} string \N{SMILEY FACE}\n";
```

Either of the above forms of "layer" specifications can be made the default for a lexical scope with the "use open ..." pragma. See open.

Once a handle is open, its layers can be altered using "binmode".

Without any such configuration, or if Perl itself is built using the system's own IO, then write operations assume that the file handle accepts only bytes and will "die" if a character larger than 255 is written to the handle. When reading, each octet from the handle becomes a byte-in-a-character. Note that this default is the same behaviour as bytes-only languages (including Perl before v5.6) would have, and is sufficient to handle native 8-bit encodings e.g. iso-8859-1, EBCDIC etc. and any legacy mechanisms for handling other encodings and binary data.

In other cases, it is the program's responsibility to transform characters into bytes using the API above before doing writes, and to transform the bytes read from a handle into characters before doing "character operations" (e.g. "lc", "/\W+/", ...).

You can also use PerlIO to convert larger amounts of data you don't want to bring into memory. For example, to convert between ISO-8859-1 (Latin 1) and UTF-8 (or UTF-EBCDIC in EBCDIC machines):

```
open(F, "<:encoding(iso-8859-1)", "data.txt") or die $!;
open(G, ">:utf8",                 "data.utf") or die $!;
while (<F>) { print G }

# Could also do "print G <F>" but that would pull
# the whole file into memory just to write it out again.
```

More examples:

```
open(my $f, "<:encoding(cp1252)")
open(my $g, ">:encoding(iso-8859-2)")
open(my $h, ">:encoding(latin9)")       # iso-8859-15
```

See also encoding for how to change the default encoding of the data in your script.

How does it work?

Here is a crude diagram of how filehandle, PerlIO, and Encode interact.

```
 filehandle <-> PerlIO       PerlIO <-> scalar (read/printed)
                   \      /
                    Encode
```

When PerlIO receives data from either direction, it fills a buffer (currently with 1024 bytes) and passes the buffer to Encode. Encode tries to convert the valid part and passes it back to PerlIO, leaving invalid parts (usually a partial character) in the buffer.
PerlIO then appends more data to the buffer, calls Encode again, and so on until the data stream ends.

To do so, PerlIO always calls (de|en)code methods with CHECK set to 1. This ensures that the method stops at the right place when it encounters partial character. The following is what happens when PerlIO and Encode tries to encode (from utf8) more than 1024 bytes and the buffer boundary happens to be in the middle of a character.

```
  A   B   C   ....  ~    \x{3000}    ....
 41  42  43   ....  7E   e3   80   80  ....
 <- buffer ---------------->
 << encoded >>>>>>>>>>
                   <- next buffer ------
```

Encode converts from the beginning to \x7E, leaving \xe3 in the buffer because it is invalid (partial character).

Unfortunately, this scheme does not work well with escape-based encodings such as ISO-2022-JP.

Line Buffering

Now let's see what happens when you try to decode from ISO-2022-JP and the buffer ends in the middle of a character.

```
          JIS208-ESC   \x{5f3e}
 A   B   C   ....  ~   \e   $   B  |DAN | ....
41  42  43   ....  7E   1b  24  41  43  46 ....
<- buffer --------------------------->
<< encoded >>>>>>>>>>>>>>>>>>>>>>>>>
```

As you see, the next buffer begins with \x43. But \x43 is 'C' in ASCII, which is wrong in this case because we are now in JISX 0208 area so it has to convert \x43\x46, not \x43. Unlike utf8 and EUC, in escape-based encodings you can't tell if a given octet is a whole character or just part of it.

Fortunately PerlIO also supports line buffer if you tell PerlIO to use one instead of fixed buffer. Since ISO-2022-JP is guaranteed to revert to ASCII at the end of the line, partial character will never happen when line buffer is used.

To tell PerlIO to use line buffer, implement ->needs_lines method for your encoding object. See Encode::Encoding for details.

Thanks to these efforts most encodings that come with Encode support PerlIO but that still leaves following encodings.

- iso-2022-kr
- MIME-B
- MIME-Header
- MIME-Q

Fortunately iso-2022-kr is hardly used (according to Jungshik) and MIME-* are very unlikely to be fed to PerlIO because they are for mail headers. See Encode::MIME::Header for details.

How can I tell whether my encoding fully supports PerlIO ?

As of this writing, any encoding whose class belongs to Encode::XS and Encode::Unicode works. The Encode module has a "perlio_ok" method which you can use before applying PerlIO encoding to the filehandle. Here is an example:

```
my $use_perlio = perlio_ok($enc);
my $layer = $use_perlio ? "<:raw" : "<:encoding($enc)";
open my $fh, $layer, $file or die "$file : $!";
while(<$fh>){
  $_ = decode($enc, $_) unless $use_perlio;
  # ....
}
```

SEE ALSO

Encode::Encoding, Encode::Supported, Encode::PerlIO, encoding, perlebcdic, "open" in perlfunc, perlunicode, utf8, the Perl Unicode Mailing List <perl-unicode@perl.org>