NAME

Encode - character encodings in Perl

SYNOPSIS

```
use Encode qw(decode encode);
$characters = decode('UTF-8', $octets, Encode::FB_CROAK);
$octets = encode('UTF-8', $characters, Encode::FB_CROAK);
```

Table of Contents

Encode consists of a collection of modules whose details are too extensive to fit in one document. This one itself explains the top-level APIs and general topics at a glance. For other topics and more details, see the documentation for these modules:

Encode::Alias - Alias definitions to encodings
Encode::Encoding - Encode Implementation Base Class
Encode::Supported - List of Supported Encodings
Encode::CN - Simplified Chinese Encodings
Encode::JP - Japanese Encodings
Encode::KR - Korean Encodings
Encode::TW - Traditional Chinese Encodings

DESCRIPTION

The `Encode` module provides the interface between Perl strings and the rest of the system. Perl strings are sequences of *characters*.

The repertoire of characters that Perl can represent is a superset of those defined by the Unicode Consortium. On most platforms the ordinal values of a character as returned by `ord(S)` is the *Unicode codepoint* for that character. The exceptions are platforms where the legacy encoding is some variant of EBCDIC rather than a superset of ASCII; see perlebcdic.

During recent history, data is moved around a computer in 8-bit chunks, often called ''bytes'' but also known as ''octets'' in standards documents. Perl is widely used to manipulate data of many types: not only strings of characters representing human or computer languages, but also ''binary'' data, being the machine's representation of numbers, pixels in an image, or just about anything.

When Perl is processing ''binary data'', the programmer wants Perl to process ''sequences of bytes''. This is not a problem for Perl: because a byte has 256 possible values, it easily fits in Perl's much larger ''logical character''.

This document mostly explains the *how*. perlunitut(1) and perlunifaq(1) explain the *why*.

TERMINOLOGY

character

A character in the range 0 .. 2**32-1 (or more); what Perl's strings are made of.

byte

A character in the range 0..255; a special case of a Perl character.

octet

8 bits of data, with ordinal values 0..255; term for bytes passed to or from a non-Perl context, such as a disk file, standard I/O stream, database, command-line argument, environment variable, socket etc.

THE PERL ENCODING API

Basic methods

encode

```
$octets = encode(ENCODING, STRING[, CHECK])
```

Encodes the scalar value *STRING* from Perl's internal form into *ENCODING* and returns a sequence of octets. *ENCODING* can be either a canonical name or an alias. For encoding names

and aliases, see ''Defining Aliases''. For CHECK, see ''Handling Malformed Data''.

For example, to convert a string from Perl's internal format into ISO-8859-1, also known as Latin1:

```
$octets = encode("iso-8859-1", $string);
```

CAVEAT: When you run $octets = encode("utf8", $string), then $octets *might not be equal to* $string. Though both contain the same data, the UTF8 flag for $octets is *always* off. When you encode anything, the UTF8 flag on the result is always off, even when it contains a completely valid utf8 string. See ''The UTF8 flag'' below.

If the $string is undef, then undef is returned.

decode

```
$string = decode(ENCODING, OCTETS[, CHECK])
```

This function returns the string that results from decoding the scalar value *OCTETS*, assumed to be a sequence of octets in *ENCODING*, into Perl's internal form. The returns the resulting string. As with *encode()*, *ENCODING* can be either a canonical name or an alias. For encoding names and aliases, see ''Defining Aliases''; for *CHECK*, see ''Handling Malformed Data''.

For example, to convert ISO-8859-1 data into a string in Perl's internal format:

```
$string = decode("iso-8859-1", $octets);
```

CAVEAT: When you run $string = decode("utf8", $octets), then $string *might not be equal to* $octets. Though both contain the same data, the UTF8 flag for $string is on. See ''The UTF8 flag'' below.

If the $string is undef, then undef is returned.

find_encoding

```
[$obj =] find_encoding(ENCODING)
```

Returns the *encoding object* corresponding to *ENCODING*. Returns undef if no matching *ENCODING* is find. The returned object is what does the actual encoding or decoding.

```
$utf8 = decode($name, $bytes);
```

is in fact

```
$utf8 = do {
$obj = find_encoding($name);
croak qq(encoding "$name" not found) unless ref $obj;
$obj->decode($bytes);
};
```

with more error checking.

You can therefore save time by reusing this object as follows;

```
my $enc = find_encoding("iso-8859-1");
while(<>) {
my $utf8 = $enc->decode($_);
... # now do something with $utf8;
}
```

Besides ''decode'' and ''encode'', other methods are available as well. For instance, name() returns the canonical name of the encoding object.

```
find_encoding("latin1")->name; # iso-8859-1
```

See Encode::Encoding for details.

from_to

```
[$length =] from_to($octets, FROM_ENC, TO_ENC [, CHECK])
```

Converts *in-place* data between two encodings. The data in `$octets` must be encoded as octets and *not* as characters in Perl's internal format. For example, to convert ISO-8859-1 data into Microsoft's CP1250 encoding:

```
from_to($octets, "iso-8859-1", "cp1250");
```

and to convert it back:

```
from_to($octets, "cp1250", "iso-8859-1");
```

Because the conversion happens in place, the data to be converted cannot be a string constant: it must be a scalar variable.

`from_to()` returns the length of the converted string in octets on success, and `undef` on error.

CAVEAT: The following operations may look the same, but are not:

```
from_to($data, "iso-8859-1", "utf8"); #1
$data = decode("iso-8859-1", $data); #2
```

Both #1 and #2 make `$data` consist of a completely valid UTF-8 string, but only #2 turns the UTF8 flag on. #1 is equivalent to:

```
$data = encode("utf8", decode("iso-8859-1", $data));
```

See ''The UTF8 flag'' below.

Also note that:

```
from_to($octets, $from, $to, $check);
```

is equivalent t:o

```
$octets = encode($to, decode($from, $octets), $check);
```

Yes, it does *not* respect the `$check` during decoding. It is deliberately done that way. If you need minute control, use `decode` followed by `encode` as follows:

```
$octets = encode($to, decode($from, $octets, $check_from), $check_to);
```

encode_utf8

```
$octets = encode_utf8($string);
```

Equivalent to `$octets = encode("utf8", $string)`. The characters in `$string` are encoded in Perl's internal format, and the result is returned as a sequence of octets. Because all possible characters in Perl have a (loose, not strict) UTF-8 representation, this function cannot fail.

decode_utf8

```
$string = decode_utf8($octets [, CHECK]);
```

Equivalent to `$string = decode("utf8", $octets [, CHECK])`. The sequence of octets represented by `$octets` is decoded from UTF-8 into a sequence of logical characters. Because not all sequences of octets are valid UTF-8, it is quite possible for this function to fail. For CHECK, see ''Handling Malformed Data''.

Listing available encodings

```
use Encode;
@list = Encode->encodings();
```

Returns a list of canonical names of available encodings that have already been loaded. To get a list of all available encodings including those that have not yet been loaded, say:

```
@all_encodings = Encode->encodings(":all");
```

Or you can give the name of a specific module:

```
@with_jp = Encode->encodings("Encode::JP");
```

When `::`'' is not in the name, ''`Encode::` is assumed.

```
@ebcdic = Encode->encodings("EBCDIC");
```

To find out in detail which encodings are supported by this package, see Encode::Supported.

Defining Aliases

To add a new alias to a given encoding, use:

```
use Encode;
use Encode::Alias;
define_alias(NEWNAME => ENCODING);
```

After that, *NEWNAME* can be used as an alias for *ENCODING*. *ENCODING* may be either the name of an encoding or an *encoding object*.

Before you do that, first make sure the alias is nonexistent using `resolve_alias()`, which returns the canonical name thereof. For example:

```
Encode::resolve_alias("latin1") eq "iso-8859-1" # true
Encode::resolve_alias("iso-8859-12") # false; nonexistent
Encode::resolve_alias($name) eq $name # true if $name is canonical
```

`resolve_alias()` does not need use `Encode::Alias` it can be imported via use `Encode qw(resolve_alias)`.

See Encode::Alias for details.

Finding IANA Character Set Registry names

The canonical name of a given encoding does not necessarily agree with IANA Character Set Registry, commonly seen as `Content-Type: text/plain; charset=`*WHATEVER*. For most cases, the canonical name works, but sometimes it does not, most notably with ''utf-8-strict''.

As of `Encode` version 2.21, a new method `mime_name()` is therefore added.

```
use Encode;
my $enc = find_encoding("UTF-8");
warn $enc->name; # utf-8-strict
warn $enc->mime_name; # UTF-8
```

See also: Encode::Encoding

Encoding via PerlIO

If your perl supports `PerlIO` (which is the default), you can use a `PerlIO` layer to decode and encode directly via a filehandle. The following two examples are fully identical in functionality:

```
### Version 1 via PerlIO
open(INPUT, "< :encoding(shiftjis)", $infile)
|| die "Can't open < $infile for reading: $!";
open(OUTPUT, "> :encoding(euc-jp)", $outfile)
|| die "Can't open > $output for writing: $!";
while (<INPUT>) { # auto decodes $_
print OUTPUT; # auto encodes $_
}
close(INPUT) || die "can't close $infile: $!";
close(OUTPUT) || die "can't close $outfile: $!";

### Version 2 via from_to()
open(INPUT, "< :raw", $infile)
|| die "Can't open < $infile for reading: $!";
open(OUTPUT, "> :raw", $outfile)
|| die "Can't open > $output for writing: $!";

while (<INPUT>) {
from_to($_, "shiftjis", "euc-jp", 1); # switch encoding
```

```
print OUTPUT; # emit raw (but properly encoded) data
}
close(INPUT) || die "can't close $infile: $!";
close(OUTPUT) || die "can't close $outfile: $!";
```

In the first version above, you let the appropriate encoding layer handle the conversion. In the second, you explicitly translate from one encoding to the other.

Unfortunately, it may be that encodings are `PerlIO`-savvy. You can check to see whether your encoding is supported by `PerlIO` by invoking the `perlio_ok` method on it:

```
Encode::perlio_ok("hz"); # false
find_encoding("euc-cn")->perlio_ok; # true wherever PerlIO is available

use Encode qw(perlio_ok); # imported upon request
perlio_ok("euc-jp")
```

Fortunately, all encodings that come with `Encode` core are `PerlIO`-savvy except for `hz` and `ISO-2022-kr`. For the gory details, see Encode::Encoding and Encode::PerlIO.

Handling Malformed Data

The optional *CHECK* argument tells `Encode` what to do when encountering malformed data. Without *CHECK*, `Encode::FB_DEFAULT` (== 0) is assumed.

As of version 2.12, `Encode` supports coderef values for `CHECK`; see below.

NOTE: Not all encodings support this feature. Some encodings ignore the *CHECK* argument. For example, Encode::Unicode ignores *CHECK* and it always croaks on error.

List of *CHECK* values

FB_DEFAULT

```
I<CHECK> = Encode::FB_DEFAULT ( == 0)
```

If *CHECK* is 0, encoding and decoding replace any malformed character with a *substitution character*. When you encode, *SUBCHAR* is used. When you decode, the Unicode REPLACEMENT CHARACTER, code point U+FFFD, is used. If the data is supposed to be UTF-8, an optional lexical warning of warning category `"utf8"` is given.

FB_CROAK

```
I<CHECK> = Encode::FB_CROAK ( == 1)
```

If *CHECK* is 1, methods immediately die with an error message. Therefore, when *CHECK* is 1, you should trap exceptions with `eval{}`, unless you really want to let it `die`.

FB_QUIET

```
I<CHECK> = Encode::FB_QUIET
```

If *CHECK* is set to `Encode::FB_QUIET` encoding and decoding immediately return the portion of the data that has been processed so far when an error occurs. The data argument is overwritten with everything after that point; that is, the unprocessed portion of the data. This is handy when you have to call `decode` repeatedly in the case where your source data may contain partial multi-byte character sequences, (that is, you are reading with a fixed-width buffer). Here's some sample code to do exactly that:

```
my($buffer, $string) = ("", "");
while (read($fh, $buffer, 256, length($buffer))) {
$string .= decode($encoding, $buffer, Encode::FB_QUIET);
# $buffer now contains the unprocessed partial character
}
```

FB_WARN

```
I<CHECK> = Encode::FB_WARN
```

This is the same as `FB_QUIET` above, except that instead of being silent on errors, it issues a warning. This is handy for when you are debugging.

FB_PERLQQ FB_HTMLCREF FB_XMLCREF

perlqq mode (*CHECK* = Encode::FB_PERLQQ)
HTML charref mode (*CHECK* = Encode::FB_HTMLCREF)
XML charref mode (*CHECK* = Encode::FB_XMLCREF)

For encodings that are implemented by the `Encode::XS` module, CHECK == `Encode::FB_PERLQQ` puts `encode` and `decode` into `perlqq` fallback mode.

When you decode, `\x`*HH* is inserted for a malformed character, where *HH* is the hex representation of the octet that could not be decoded to utf8. When you encode, `\x{`*HHHH*`}` will be inserted, where *HHHH* is the Unicode code point (in any number of hex digits) of the character that cannot be found in the character repertoire of the encoding.

The HTML/XML character reference modes are about the same. In place of `\x{`*HHHH*`}`, HTML uses `&#`*NNN*`;` where *NNN* is a decimal number, and XML uses `&#x`*HHHH*`;` where *HHHH* is the hexadecimal number.

In `Encode` 2.10 or later, `LEAVE_SRC` is also implied.

The bitmask

These modes are all actually set via a bitmask. Here is how the `FB_`*XXX* constants are laid out. You can import the `FB_`*XXX* constants via `use Encode qw(:fallbacks)`, and you can import the generic bitmask constants via `use Encode qw(:fallback_all)`.

```
 FB_DEFAULT FB_CROAK FB_QUIET FB_WARN FB_PERLQQ
 DIE_ON_ERR 0x0001 X
 WARN_ON_ERR 0x0002 X
 RETURN_ON_ERR 0x0004 X X
 LEAVE_SRC 0x0008 X
 PERLQQ 0x0100 X
 HTMLCREF 0x0200
 XMLCREF 0x0400
```

LEAVE_SRC

```
 Encode::LEAVE_SRC
```

If the `Encode::LEAVE_SRC` bit is *not* set but *CHECK* is set, then the source string to *encode()* or *decode()* will be overwritten in place. If you're not interested in this, then bitwise-OR it with the bitmask.

coderef for CHECK

As of `Encode` 2.12, CHECK can also be a code reference which takes the ordinal value of the unmapped character as an argument and returns octets that represent the fallback character. For instance:

```
 $ascii = encode("ascii", $utf8, sub{ sprintf "<U+%04X>", shift });
```

Acts like `FB_PERLQQ` but U+*XXXX* is used instead of `\x{`*XXXX*`}`.

Even the fallback for `decode` must return octets, which are then decoded with the character encoding that `decode` accepts. So for example if you wish to decode octets as UTF-8, and use ISO-8859-15 as a fallback for bytes that are not valid UTF-8, you could write

```
$str = decode 'UTF-8', $octets, sub {
my $tmp = chr shift;
from_to $tmp, 'ISO-8859-15', 'UTF-8';
return $tmp;
};
```

Defining Encodings

To define a new encoding, use:

```
use Encode qw(define_encoding);
define_encoding($object, CANONICAL_NAME [, alias...]);
```

CANONICAL_NAME will be associated with *object*. The object should provide the interface described in Encode::Encoding. If more than two arguments are provided, additional arguments are considered aliases for *object*.

See Encode::Encoding for details.

The UTF8 flag

Before the introduction of Unicode support in Perl, The `eq` operator just compared the strings represented by two scalars. Beginning with Perl 5.8, `eq` compares two strings with simultaneous consideration of *the UTF8 flag*. To explain why we made it so, I quote from page 402 of *Programming Perl, 3rd ed.*

Goal #1:
   Old byte-oriented programs should not spontaneously break on the old byte-oriented data they used to work on.

Goal #2:
   Old byte-oriented programs should magically start working on the new character-oriented data when appropriate.

Goal #3:
   Programs should run just as fast in the new character-oriented mode as in the old byte-oriented mode.

Goal #4:
   Perl should remain one language, rather than forking into a byte-oriented Perl and a character-oriented Perl.

When *Programming Perl, 3rd ed.* was written, not even Perl 5.6.0 had been born yet, many features documented in the book remained unimplemented for a long time. Perl 5.8 corrected much of this, and the introduction of the UTF8 flag is one of them. You can think of there being two fundamentally different kinds of strings and string-operations in Perl: one a byte-oriented mode for when the internal UTF8 flag is off, and the other a character-oriented mode for when the internal UTF8 flag is on.

Here is how `Encode` handles the UTF8 flag.

• When you *encode*, the resulting UTF8 flag is always **off**.

• When you *decode*, the resulting UTF8 flag is **on**--*unless* you can unambiguously represent data. Here is what we mean by ''unambiguously''. After `$utf8 = decode("foo", $octet)`,

```
When $octet is... The UTF8 flag in $utf8 is
----------------------------------------------
In ASCII only (or EBCDIC only) OFF
In ISO-8859-1 ON
In any other Encoding ON
----------------------------------------------
```

As you see, there is one exception: in ASCII. That way you can assume Goal #1. And with `Encode`, Goal #2 is assumed but you still have to be careful in the cases mentioned in the

CAVEAT paragraphs above.

This UTF8 flag is not visible in Perl scripts, exactly for the same reason you cannot (or rather, you *don't have to*) see whether a scalar contains a string, an integer, or a floating-point number. But you can still peek and poke these if you will. See the next section.

Messing with Perl's Internals

The following API uses parts of Perl's internals in the current implementation. As such, they are efficient but may change in a future release.

is_utf8

```
is_utf8(STRING [, CHECK])
```

[INTERNAL] Tests whether the UTF8 flag is turned on in the *STRING*. If *CHECK* is true, also checks whether *STRING* contains well-formed UTF-8. Returns true if successful, false otherwise.

As of Perl 5.8.1, utf8 also has the `utf8::is_utf8` function.

_utf8_on

```
_utf8_on(STRING)
```

[INTERNAL] Turns the *STRING*'s internal UTF8 flag **on**. The *STRING* is *not* checked for containing only well-formed UTF-8. Do not use this unless you *know with absolute certainty* that the STRING holds only well-formed UTF-8. Returns the previous state of the UTF8 flag (so please don't treat the return value as indicating success or failure), or `undef` if *STRING* is not a string.

NOTE: For security reasons, this function does not work on tainted values.

_utf8_off

```
_utf8_off(STRING)
```

[INTERNAL] Turns the *STRING*'s internal UTF8 flag **off**. Do not use frivolously. Returns the previous state of the UTF8 flag, or `undef` if *STRING* is not a string. Do not treat the return value as indicative of success or failure, because that isn't what it means: it is only the previous setting.

NOTE: For security reasons, this function does not work on tainted values.

UTF−8 vs. utf8 vs. UTF8

```
....We now view strings not as sequences of bytes, but as sequences
of numbers in the range 0 .. 2**32-1 (or in the case of 64-bit
computers, 0 .. 2**64-1) -- Programming Perl, 3rd ed.
```

That has historically been Perl's notion of UTF-8, as that is how UTF-8 was first conceived by Ken Thompson when he invented it. However, thanks to later revisions to the applicable standards, official UTF-8 is now rather stricter than that. For example, its range is much narrower (0 .. 0x10_FFFF to cover only 21 bits instead of 32 or 64 bits) and some sequences are not allowed, like those used in surrogate pairs, the 31 non-character code points 0xFDD0 .. 0xFDEF, the last two code points in *any* plane (0x*XX*_FFFE and 0x*XX*_FFFF), all non-shortest encodings, etc.

The former default in which Perl would always use a loose interpretation of UTF-8 has now been overruled:

```
From: Larry Wall <larry@wall.org>
Date: December 04, 2004 11:51:58 JST
To: perl-unicode@perl.org
Subject: Re: Make Encode.pm support the real UTF-8
Message-Id: <20041204025158.GA28754@wall.org>

On Fri, Dec 03, 2004 at 10:12:12PM +0000, Tim Bunce wrote:
: I've no problem with 'utf8' being perl's unrestricted uft8 encoding,
: but "UTF-8" is the name of the standard and should give the
: corresponding behaviour.
```

```
For what it's worth, that's how I've always kept them straight in my
head.

Also for what it's worth, Perl 6 will mostly default to strict but
make it easy to switch back to lax.

Larry
```

Got that? As of Perl 5.8.7, **"UTF-8"** means UTF-8 in its current sense, which is conservative and strict and security-conscious, whereas **"utf8"** means UTF-8 in its former sense, which was liberal and loose and lax. `Encode` version 2.10 or later thus groks this subtle but critically important distinction between `"UTF-8"` and `"utf8"`.

```
encode("utf8", "\x{FFFF_FFFF}", 1); # okay
encode("UTF-8", "\x{FFFF_FFFF}", 1); # croaks
```

In the `Encode` module, `"UTF-8"` is actually a canonical name for `"utf-8-strict"`. That hyphen between the `"UTF"` and the `"8"` is critical; without it, `Encode` goes ''liberal'' and (perhaps overly-)permissive:

```
find_encoding("UTF-8")->name # is 'utf-8-strict'
find_encoding("utf-8")->name # ditto. names are case insensitive
find_encoding("utf_8")->name # ditto. "_" are treated as "-"
find_encoding("UTF8")->name # is 'utf8'.
```

Perl's internal UTF8 flag is called ''UTF8'', without a hyphen. It indicates whether a string is internally encoded as ''utf8'', also without a hyphen.

SEE ALSO

Encode::Encoding, Encode::Supported, Encode::PerlIO, encoding, perlebcdic(1), ''open'' in perlfunc(1), perlunicode(1), perluniintro(1), perlunifaq(1), perlunitut(1) utf8, the Perl Unicode Mailing List <http://lists.perl.org/list/perl-unicode.html>

MAINTAINER

This project was originated by the late Nick Ing-Simmons and later maintained by Dan Kogai <*dankogai@cpan.org*>. See AUTHORS for a full list of people involved. For any questions, send mail to <*perl-unicode@perl.org*> so that we can all share.

While Dan Kogai retains the copyright as a maintainer, credit should go to all those involved. See AUTHORS for a list of those who submitted code to the project.

COPYRIGHT

Copyright 2002-2013 Dan Kogai <*dankogai@cpan.org*>.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.