NAME

Encode::Encoding - Encode Implementation Base Class

SYNOPSIS

```
package Encode::MyEncoding;
use base qw(Encode::Encoding);

__PACKAGE__->Define(qw(myCanonical myAlias));
```

DESCRIPTION

As mentioned in *Encode*, encodings are (in the current implementation at least) defined as objects. The mapping of encoding name to object is via the `%Encode::Encoding` hash. Though you can directly manipulate this hash, it is strongly encouraged to use this base class module and add encode() and decode() methods.

Methods you should implement

You are strongly encouraged to implement methods below, at least either encode() or decode().

->encode($string [,$check])

MUST return the octet sequence representing *$string*.

- If *$check* is true, it SHOULD modify *$string* in place to remove the converted part (i.e. the whole string unless there is an error). If perlio_ok() is true, SHOULD becomes MUST.
- If an error occurs, it SHOULD return the octet sequence for the fragment of string that has been converted and modify $string in-place to remove the converted part leaving it starting with the problem fragment. If perlio_ok() is true, SHOULD becomes MUST.
- If *$check* is is false then `encode` MUST make a "best effort" to convert the string - for example, by using a replacement character.

->decode($octets [,$check])

MUST return the string that *$octets* represents.

- If *$check* is true, it SHOULD modify *$octets* in place to remove the converted part (i.e. the whole sequence unless there is an error). If perlio_ok() is true, SHOULD becomes MUST.
- If an error occurs, it SHOULD return the fragment of string that has been converted and modify $octets in-place to remove the converted part leaving it starting with the problem fragment. If perlio_ok() is true, SHOULD becomes MUST.
- If *$check* is false then `decode` should make a "best effort" to convert the string - for example by using Unicode's "\x{FFFD}" as a replacement character.

If you want your encoding to work with *encoding* pragma, you should also implement the method below.

->cat_decode($destination, $octets, $offset, $terminator [,$check])

MUST decode *$octets* with *$offset* and concatenate it to *$destination*. Decoding will terminate when $terminator (a string) appears in output. *$offset* will be modified to the last $octets position at end of decode. Returns true if $terminator appears output, else returns false.

Other methods defined in Encode::Encodings

You do not have to override methods shown below unless you have to.

->name

Predefined As:

```
sub name  { return shift->{'Name'} }
```

MUST return the string representing the canonical name of the encoding.

->mime_name

Predefined As:

```
sub mime_name{
  require Encode::MIME::Name;
  return Encode::MIME::Name::get_mime_name(shift->name);
}
```

MUST return the string representing the IANA charset name of the encoding.

->renew

Predefined As:

```
sub renew {
  my $self = shift;
  my $clone = bless { %$self } => ref($self);
  $clone->{renewed}++;
  return $clone;
}
```

This method reconstructs the encoding object if necessary. If you need to store the state during encoding, this is where you clone your object.

PerlIO ALWAYS calls this method to make sure it has its own private encoding object.

->renewed

Predefined As:

```
sub renewed { $_[0]->{renewed} || 0 }
```

Tells whether the object is renewed (and how many times). Some modules emit `Use of uninitialized value in null operation` warning unless the value is numeric so return 0 for false.

->perlio_ok()

Predefined As:

```
sub perlio_ok {
    eval{ require PerlIO::encoding };
    return $@ ? 0 : 1;
}
```

If your encoding does not support PerlIO for some reasons, just;

```
sub perlio_ok { 0 }
```

->needs_lines()

Predefined As:

```
sub needs_lines { 0 };
```

If your encoding can work with PerlIO but needs line buffering, you MUST define this method so it returns true. 7bit ISO-2022 encodings are one example that needs this. When this method is missing, false is assumed.

Example: Encode::ROT13

```
package Encode::ROT13;
use strict;
use base qw(Encode::Encoding);

__PACKAGE__->Define('rot13');

sub encode($$;$){
    my ($obj, $str, $chk) = @_;
    $str =~ tr/A-Za-z/N-ZA-Mn-za-m/;
    $_[1] = '' if $chk; # this is what in-place edit means
    return $str;
}

# Jr pna or ynml yvxr guvf;
*decode = \&encode;

1;
```

Why the heck Encode API is different?

It should be noted that the *$check* behaviour is different from the outer public API. The logic is that the "unchecked" case is useful when the encoding is part of a stream which may be reporting errors (e.g. STDERR). In such cases, it is desirable to get everything through somehow without causing additional errors which obscure the original one. Also, the encoding is best placed to know what the correct replacement character is, so if that is the desired behaviour then letting low level code do it is the most efficient.

By contrast, if *$check* is true, the scheme above allows the encoding to do as much as it can and tell the layer above how much that was. What is lacking at present is a mechanism to report what went wrong. The most likely interface will be an additional method call to the object, or perhaps (to avoid forcing per-stream objects on otherwise stateless encodings) an additional parameter.

It is also highly desirable that encoding classes inherit from `Encode::Encoding` as a base class. This allows that class to define additional behaviour for all encoding objects.

```
package Encode::MyEncoding;
use base qw(Encode::Encoding);

__PACKAGE__->Define(qw(myCanonical myAlias));
```

to create an object with `bless {Name => ...}, $class`, and call define_encoding. They inherit their `name` method from `Encode::Encoding`.

Compiled Encodings

For the sake of speed and efficiency, most of the encodings are now supported via a *compiled form*: XS modules generated from UCM files. Encode provides the enc2xs tool to achieve that. Please see *enc2xs* for more details.

SEE ALSO

perlmod, *enc2xs*

Scheme 1

> The fixup routine gets passed the remaining fragment of string being processed. It modifies it in place to remove bytes/characters it can understand and returns a string used to represent

them. For example:

```
sub fixup {
  my $ch = substr($_[0],0,1,'');
  return sprintf("\x{%02X}",ord($ch);
}
```

This scheme is close to how the underlying C code for Encode works, but gives the fixup routine very little context.

Scheme 2

The fixup routine gets passed the original string, an index into it of the problem area, and the output string so far. It appends what it wants to the output string and returns a new index into the original string. For example:

```
sub fixup {
  # my ($s,$i,$d) = @_;
  my $ch = substr($_[0],$_[1],1);
  $_[2] .= sprintf("\x{%02X}",ord($ch);
  return $_[1]+1;
}
```

This scheme gives maximal control to the fixup routine but is more complicated to code, and may require that the internals of Encode be tweaked to keep the original string intact.

Other Schemes

Hybrids of the above.

Multiple return values rather than in-place modifications.

Index into the string could be `pos($str)` allowing `s/\G...//`.