NAME

PerlIO - On demand loader for PerlIO layers and root of PerlIO::* name space

SYNOPSIS

```
# support platform-native and CRLF text files
open(my $fh, "<:crlf", "my.txt") or die "open failed: $!";

# append UTF-8 encoded text
open(my $fh, ">>:encoding(UTF-8)", "some.log")
  or die "open failed: $!";

# portably open a binary file for reading
open(my $fh, "<", "his.jpg") or die "open failed: $!";
binmode($fh) or die "binmode failed: $!";

Shell:
  PERLIO=:perlio perl ....
```

DESCRIPTION

When an undefined layer ’foo’ is encountered in an "open" or "binmode" layer specification then C code performs the equivalent of:

```
use PerlIO ’foo’;
```

The Perl code in PerlIO.pm then attempts to locate a layer by doing

```
require PerlIO::foo;
```

Otherwise the "PerlIO" package is a place holder for additional PerlIO related functions.

Layers

Generally speaking, PerlIO layers (previously sometimes referred to as "disciplines") are an ordered stack applied to a filehandle (specified as a space- or colon-separated list, conventionally written with a leading colon). Each layer performs some operation on any input or output, except when bypassed such as with "sysread" or "syswrite". Read operations go through the stack in the order they are set (left to right), and write operations in the reverse order.

There are also layers which actually just set flags on lower layers, or layers that modify the current stack but don’t persist on the stack themselves; these are referred to as pseudo-layers.

When opening a handle, it will be opened with any layers specified explicitly in the **open()** call (or the platform defaults, if specified as a colon with no following layers).

If layers are not explicitly specified, the handle will be opened with the layers specified by the ${^OPEN} variable (usually set by using the open pragma for a lexical scope, or the "-C" command-line switch or "PERL_UNICODE" environment variable for the main program scope).

If layers are not specified in the **open()** call or "${^OPEN}" variable, the handle will be opened with the default layer stack configured for that architecture; see "Defaults and how to override them".

Some layers will automatically insert required lower level layers if not present; for example ":perlio" will insert ":unix" below itself for low level IO, and ":encoding" will insert the platform defaults for buffered IO.

The "binmode" function can be called on an opened handle to push additional layers onto the stack, which may also modify the existing layers. "binmode" called with no layers will remove or unset any existing layers which transform the byte stream, making the handle suitable for binary data.

The following layers are currently defined:

:unix
: Lowest level layer which provides basic PerlIO operations in terms of UNIX/POSIX numeric file descriptor calls (**open()**, **read()**, **write()**, **lseek()**, **close()**). It is used even on non-Unix architectures, and most other layers operate on top of it.

:stdio
: Layer which calls "fread", "fwrite" and "fseek"/"ftell" etc. Note that as this is "real" stdio it will ignore any layers beneath it and go straight to the operating system via the C library as usual. This layer implements both low level IO and buffering, but is rarely used on modern architectures.

:perlio
: A from scratch implementation of buffering for PerlIO. Provides fast access to the buffer for "sv_gets" which implements Perl's readline/<> and in general attempts to minimize data copying.

  ":perlio" will insert a ":unix" layer below itself to do low level IO.

:crlf
: A layer that implements DOS/Windows like CRLF line endings. On read converts pairs of CR,LF to a single "\n" newline character. On write converts each "\n" to a CR,LF pair. Note that this layer will silently refuse to be pushed on top of itself.

It currently does *not* mimic MS-DOS as far as treating of Control-Z as being an end-of-file marker.

On DOS/Windows like architectures where this layer is part of the defaults, it also acts like the ":perlio" layer, and removing the CRLF translation (such as with ":raw") will only unset the CRLF translation flag. Since Perl 5.14, you can also apply another ":crlf" layer later, such as when the CRLF translation must occur after an encoding layer. On other architectures, it is a mundane CRLF translation layer and can be added and removed normally.

```
# translate CRLF after encoding on Perl 5.14 or newer
binmode $fh, ":raw:encoding(UTF-16LE):crlf"
  or die "binmode failed: $!";
```

:utf8

Pseudo-layer that declares that the stream accepts Perl's *internal* upgraded encoding of characters, which is approximately UTF-8 on ASCII machines, but UTF-EBCDIC on EBCDIC machines. This allows any character Perl can represent to be read from or written to the stream.

This layer (which actually sets a flag on the preceding layer, and is implicitly set by any ":encoding" layer) does not translate or validate byte sequences. It instead indicates that the byte stream will have been arranged by other layers to be provided in Perl's internal upgraded encoding, which Perl code (and correctly written XS code) will interpret as decoded Unicode characters.

CAUTION: Do not use this layer to translate from UTF-8 bytes, as invalid UTF-8 or binary data will result in malformed Perl strings. It is unlikely to produce invalid UTF-8 when used for output, though it will instead produce UTF-EBCDIC on EBCDIC systems. The ":encoding(UTF-8)" layer (hyphen is significant) is preferred as it will ensure translation between valid UTF-8 bytes and valid Unicode characters.

:bytes

This is the inverse of the ":utf8" pseudo-layer. It turns off the flag on the layer below so that data read from it is considered to be Perl's internal downgraded encoding, thus interpreted as the native single-byte encoding of Latin-1 or EBCDIC. Likewise on output Perl will warn if a "wide" character (a codepoint not in the range 0..255) is written to a such a stream.

This is very dangerous to push on a handle using an ":encoding" layer, as such a layer assumes to be working with Perl's internal upgraded encoding, so you will likely get a mangled result. Instead use ":raw" or ":pop" to remove encoding layers.

:raw

The ":raw" pseudo-layer is *defined* as being identical to calling "binmode($fh)" - the stream is made suitable for passing binary data, i.e. each byte is passed as-is. The stream will still be buffered (but this was not always true before Perl 5.14).

In Perl 5.6 and some books the ":raw" layer is documented as the inverse of the ":crlf" layer. That is no longer the case - other layers which would alter the binary nature of the stream are also disabled. If you want UNIX line endings on a platform that normally does CRLF translation, but still want UTF-8 or encoding defaults, the appropriate thing to do is to add ":perlio" to the PERLIO environment variable, or open the handle explicitly with that layer, to replace the platform default of ":crlf".

The implementation of ":raw" is as a pseudo-layer which when "pushed" pops itself and then any layers which would modify the binary data stream. (Undoing ":utf8" and ":crlf" may be implemented by clearing flags rather than popping layers but that is an implementation detail.)

As a consequence of the fact that ":raw" normally pops layers, it usually only makes sense to have it as the only or first element in a layer specification. When used as the first element it provides a known base on which to build e.g.

```
open(my $fh,">:raw:encoding(UTF-8)",...)
  or die "open failed: $!";
```

will construct a "binary" stream regardless of the platform defaults, but then enable UTF-8 translation.

:pop

A pseudo-layer that removes the top-most layer. Gives Perl code a way to manipulate the layer stack. Note that ":pop" only works on real layers and will not undo the effects of pseudo-layers or flags like ":utf8". An example of a possible use might be:

```
open(my $fh,...) or die "open failed: $!";
...
binmode($fh,":encoding(...)") or die "binmode failed: $!";
# next chunk is encoded
...
binmode($fh,":pop") or die "binmode failed: $!";
# back to un-encoded
```

A more elegant (and safer) interface is needed.

Custom Layers

It is possible to write custom layers in addition to the above builtin ones, both in C/XS and Perl, as a module named "PerlIO::<layer name>". Some custom layers come with the Perl distribution.

:encoding
: Use ":encoding(ENCODING)" to transparently do character set and encoding transformations, for example from Shift-JIS to Unicode. Note that an ":encoding" also enables ":utf8". See PerlIO::encoding for more information.

:mmap
: A layer which implements "reading" of files by using "mmap()" to make a (whole) file appear in the process's address space, and then using that as PerlIO's "buffer". This *may* be faster in certain circumstances for large files, and may result in less physical memory use when multiple processes are reading the same file.

  Files which are not "mmap()"-able revert to behaving like the ":perlio" layer. Writes also behave like the ":perlio" layer, as "mmap()" for write needs extra house-keeping (to extend the file) which negates any advantage.

  The ":mmap" layer will not exist if the platform does not support "mmap()". See PerlIO::mmap for more information.

:via ":via(MODULE)" allows a transformation to be applied by an arbitrary Perl module, for example compression / decompression, encryption / decryption. See PerlIO::via for more information.

:scalar
: A layer implementing "in memory" files using scalar variables, automatically used in place of the platform defaults for IO when opening such a handle. As such, the scalar is expected to act like a file, only containing or storing bytes. See PerlIO::scalar for more information.

Alternatives to raw

To get a binary stream an alternate method is to use:

```
open(my $fh,"<","whatever") or die "open failed: $!";
binmode($fh) or die "binmode failed: $!";
```

This has the advantage of being backward compatible with older versions of Perl that did not use PerlIO or where ":raw" was buggy (as it was before Perl 5.14).

To get an unbuffered stream specify an unbuffered layer (e.g. ":unix") in the open call:

```
    open(my $fh,"<:unix",$path) or die "open failed: $!";
```

Defaults and how to override them

If the platform is MS-DOS like and normally does CRLF to "\n" translation for text files then the default layers are:

```
  :unix:crlf
```

Otherwise if "Configure" found out how to do "fast" IO using the system's stdio (not common on modern architectures), then the default layers are:

```
  :stdio
```

Otherwise the default layers are

```
  :unix:perlio
```

Note that the "default stack" depends on the operating system and on the Perl version, and both the compile-time and runtime configurations of Perl. The default can be overridden by setting the environment variable PERLIO to a space or colon separated list of layers, however this cannot be used to set layers that require loading modules like ":encoding".

This can be used to see the effect of/bugs in the various layers e.g.

```
  cd .../perl/t
  PERLIO=:stdio  ./perl harness
  PERLIO=:perlio ./perl harness
```

For the various values of PERLIO see "PERLIO" in perlrun.

The following table summarizes the default layers on UNIX-like and DOS-like platforms and depending on the setting of $ENV{PERLIO}:

```
 PERLIO     UNIX-like                   DOS-like
 ------     ---------                   --------
 unset / "" :unix:perlio / :stdio [1]   :unix:crlf
 :stdio     :stdio                      :stdio
 :perlio    :unix:perlio                :unix:perlio

 # [1] ":stdio" if Configure found out how to do "fast stdio" (depends
 # on the stdio implementation) and in Perl 5.8, else ":unix:perlio"
```

Querying the layers of filehandles

The following returns the **names** of the PerlIO layers on a filehandle.

```
   my @layers = PerlIO::get_layers($fh); # Or FH, *FH, "FH".
```

The layers are returned in the order an **open()** or **binmode()** call would use them, and without colons.

By default the layers from the input side of the filehandle are returned; to get the output side, use the optional "output" argument:

```
   my @layers = PerlIO::get_layers($fh, output => 1);
```

(Usually the layers are identical on either side of a filehandle but for example with sockets there may be differences.)

There is no **set_layers()**, nor does **get_layers()** return a tied array mirroring the stack, or anything fancy like that. This is not accidental or unintentional. The PerlIO layer stack is a bit more complicated than just a stack (see for example the behaviour of ":raw"). You are supposed to use **open()** and **binmode()** to manipulate the stack.

Implementation details follow, please close your eyes.

The arguments to layers are by default returned in parentheses after the name of the layer, and certain layers (like ":utf8") are not real layers but instead flags on real layers; to get all of these returned separately, use the optional "details" argument:

```
   my @layer_and_args_and_flags = PerlIO::get_layers($fh, details => 1);
```

The result will be up to be three times the number of layers: the first element will be a name, the second element the arguments (unspecified arguments will be "undef"), the third element the flags, the fourth element a name again, and so forth.

You may open your eyes now.

AUTHOR

Nick Ing-Simmons <nick@ing-simmons.net>

SEE ALSO

"binmode" in perlfunc, "open" in perlfunc, perlunicode, perliol, Encode