NAME

perliol - C API for Perl's implementation of IO in Layers.

SYNOPSIS

```
/* Defining a layer ... */
#include <perliol.h>
```

DESCRIPTION

This document describes the behavior and implementation of the PerlIO abstraction described in *perlapio* when `USE_PERLIO` is defined (and `USE_SFIO` is not).

History and Background

The PerlIO abstraction was introduced in perl5.003_02 but languished as just an abstraction until perl5.7.0. However during that time a number of perl extensions switched to using it, so the API is mostly fixed to maintain (source) compatibility.

The aim of the implementation is to provide the PerlIO API in a flexible and platform neutral manner. It is also a trial of an "Object Oriented C, with vtables" approach which may be applied to Perl 6.

Basic Structure

PerlIO is a stack of layers.

The low levels of the stack work with the low-level operating system calls (file descriptors in C) getting bytes in and out, the higher layers of the stack buffer, filter, and otherwise manipulate the I/O, and return characters (or bytes) to Perl. Terms *above* and *below* are used to refer to the relative positioning of the stack layers.

A layer contains a "vtable", the table of I/O operations (at C level a table of function pointers), and status flags. The functions in the vtable implement operations like "open", "read", and "write".

When I/O, for example "read", is requested, the request goes from Perl first down the stack using "read" functions of each layer, then at the bottom the input is requested from the operating system services, then the result is returned up the stack, finally being interpreted as Perl data.

The requests do not necessarily go always all the way down to the operating system: that's where PerlIO buffering comes into play.

When you do an open() and specify extra PerlIO layers to be deployed, the layers you specify are "pushed" on top of the already existing default stack. One way to see it is that "operating system is on the left" and "Perl is on the right".

What exact layers are in this default stack depends on a lot of things: your operating system, Perl version, Perl compile time configuration, and Perl runtime configuration. See *PerlIO*, *"PERLIO" in perlrun*, and *open* for more information.

binmode() operates similarly to open(): by default the specified layers are pushed on top of the existing stack.

However, note that even as the specified layers are "pushed on top" for open() and binmode(), this doesn't mean that the effects are limited to the "top": PerlIO layers can be very 'active' and inspect and affect layers also deeper in the stack. As an example there is a layer called "raw" which repeatedly "pops" layers until it reaches the first layer that has declared itself capable of handling binary data. The "pushed" layers are processed in left-to-right order.

sysopen() operates (unsurprisingly) at a lower level in the stack than open(). For example in Unix or Unix-like systems sysopen() operates directly at the level of file descriptors: in the terms of PerlIO layers, it uses only the "unix" layer, which is a rather thin wrapper on top of the Unix file descriptors.

Layers vs Disciplines

Initial discussion of the ability to modify IO streams behaviour used the term "discipline" for the entities which were added. This came (I believe) from the use of the term in "sfio", which in turn borrowed it from "line disciplines" on Unix terminals. However, this document (and the C code) uses the term "layer".

This is, I hope, a natural term given the implementation, and should avoid connotations that are inherent in earlier uses of "discipline" for things which are rather different.

Data Structures

The basic data structure is a PerlIOl:

```
typedef struct _PerlIO PerlIOl;
typedef struct _PerlIO_funcs PerlIO_funcs;
typedef PerlIOl *PerlIO;

struct _PerlIO
{
 PerlIOl * next;       /* Lower layer */
 PerlIO_funcs * tab;        /* Functions for this layer */
 IV  flags;      /* Various flags for state */
};
```

A `PerlIOl *` is a pointer to the struct, and the *application* level `PerlIO *` is a pointer to a `PerlIOl *` - i.e. a pointer to a pointer to the struct. This allows the application level `PerlIO *` to remain constant while the actual `PerlIOl *` underneath changes. (Compare perl's SV `*` which remains constant while its `sv_any` field changes as the scalar's type changes.) An IO stream is then in general represented as a pointer to this linked-list of "layers".

It should be noted that because of the double indirection in a `PerlIO *`, a `&(perlio->next)` "is" a `PerlIO *`, and so to some degree at least one layer can use the "standard" API on the next layer down.

A "layer" is composed of two parts:

1. The functions and attributes of the "layer class".
2. The per-instance data for a particular handle.

Functions and Attributes

The functions and attributes are accessed via the "tab" (for table) member of `PerlIOl`. The functions (methods of the layer "class") are fixed, and are defined by the `PerlIO_funcs` type. They are broadly the same as the public `PerlIO_xxxxx` functions:

```
  struct _PerlIO_funcs
  {
   Size_t  fsize;
   char *  name;
   Size_t  size;
   IV  kind;
   IV  (*Pushed)(pTHX_ PerlIO *f,const char *mode,SV *arg, PerlIO_funcs
*tab);
   IV  (*Popped)(pTHX_ PerlIO *f);
   PerlIO * (*Open)(pTHX_ PerlIO_funcs *tab,
     PerlIO_list_t *layers, IV n,
     const char *mode,
     int fd, int imode, int perm,
```

```
      PerlIO *old,
      int narg, SV **args);
    IV  (*Binmode)(pTHX_ PerlIO *f);
    SV *  (*Getarg)(pTHX_ PerlIO *f, CLONE_PARAMS *param, int flags)
    IV  (*Fileno)(pTHX_ PerlIO *f);
    PerlIO *     (*Dup)(pTHX_ PerlIO *f, PerlIO *o, CLONE_PARAMS *param, int
flags)
    /* Unix-like functions - cf sfio line disciplines */
    SSize_t (*Read)(pTHX_ PerlIO *f, void *vbuf, Size_t count);
    SSize_t (*Unread)(pTHX_ PerlIO *f, const void *vbuf, Size_t count);
    SSize_t (*Write)(pTHX_ PerlIO *f, const void *vbuf, Size_t count);
    IV  (*Seek)(pTHX_ PerlIO *f, Off_t offset, int whence);
    Off_t (*Tell)(pTHX_ PerlIO *f);
    IV  (*Close)(pTHX_ PerlIO *f);
    /* Stdio-like buffered IO functions */
    IV  (*Flush)(pTHX_ PerlIO *f);
    IV  (*Fill)(pTHX_ PerlIO *f);
    IV  (*Eof)(pTHX_ PerlIO *f);
    IV  (*Error)(pTHX_ PerlIO *f);
    void  (*Clearerr)(pTHX_ PerlIO *f);
    void  (*Setlinebuf)(pTHX_ PerlIO *f);
    /* Perl's snooping functions */
    STDCHAR * (*Get_base)(pTHX_ PerlIO *f);
    Size_t (*Get_bufsiz)(pTHX_ PerlIO *f);
    STDCHAR * (*Get_ptr)(pTHX_ PerlIO *f);
    SSize_t (*Get_cnt)(pTHX_ PerlIO *f);
    void  (*Set_ptrcnt)(pTHX_ PerlIO *f,STDCHAR *ptr,SSize_t cnt);
   };
```

The first few members of the struct give a function table size for compatibility check "name" for the layer, the size to `malloc` for the per-instance data, and some flags which are attributes of the class as whole (such as whether it is a buffering layer), then follow the functions which fall into four basic groups:

1. Opening and setup functions
2. Basic IO operations
3. Stdio class buffering options.
4. Functions to support Perl's traditional "fast" access to the buffer.

A layer does not have to implement all the functions, but the whole table has to be present. Unimplemented slots can be NULL (which will result in an error when called) or can be filled in with stubs to "inherit" behaviour from a "base class". This "inheritance" is fixed for all instances of the layer, but as the layer chooses which stubs to populate the table, limited "multiple inheritance" is possible.

Per-instance Data

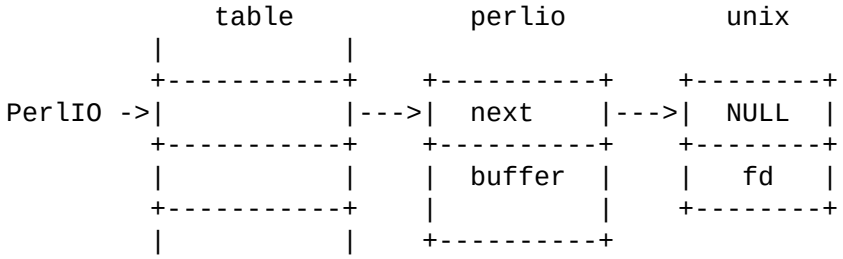
The per-instance data are held in memory beyond the basic PerlIOl struct, by making a PerlIOl the first member of the layer's struct thus:

```
 typedef struct
 {
  struct _PerlIO base;       /* Base "class" info */
  STDCHAR * buf;        /* Start of buffer */
  STDCHAR * end;        /* End of valid part of buffer */
  STDCHAR * ptr;        /* Current position in buffer */
```

```
 Off_t  posn;       /* Offset of buf into the file */
 Size_t  bufsiz;     /* Real size of buffer */
 IV  oneword;    /* Emergency buffer */
} PerlIOBuf;
```

In this way (as for perl's scalars) a pointer to a PerlIOBuf can be treated as a pointer to a PerlIOl.

Layers in action.



The above attempts to show how the layer scheme works in a simple case. The application's `PerlIO *` points to an entry in the table(s) representing open (allocated) handles. For example the first three slots in the table correspond to `stdin`,`stdout` and `stderr`. The table in turn points to the current "top" layer for the handle - in this case an instance of the generic buffering layer "perlio". That layer in turn points to the next layer down - in this case the low-level "unix" layer.

The above is roughly equivalent to a "stdio" buffered stream, but with much more flexibility:

- If Unix level `read`/`write`/`lseek` is not appropriate for (say) sockets then the "unix" layer can be replaced (at open time or even dynamically) with a "socket" layer.
- Different handles can have different buffering schemes. The "top" layer could be the "mmap" layer if reading disk files was quicker using `mmap` than `read`. An "unbuffered" stream can be implemented simply by not having a buffer layer.
- Extra layers can be inserted to process the data as it flows through. This was the driving need for including the scheme in perl 5.7.0+ - we needed a mechanism to allow data to be translated between perl's internal encoding (conceptually at least Unicode as UTF-8), and the "native" format used by the system. This is provided by the ":encoding(xxxx)" layer which typically sits above the buffering layer.
- A layer can be added that does "\n" to CRLF translation. This layer can be used on any platform, not just those that normally do such things.

Per-instance flag bits

The generic flag bits are a hybrid of `O_XXXXX` style flags deduced from the mode string passed to `PerlIO_open()`, and state bits for typical buffer layers.

PERLIO_F_EOF

End of file.

PERLIO_F_CANWRITE

Writes are permitted, i.e. opened as "w" or "r+" or "a", etc.

PERLIO_F_CANREAD

Reads are permitted i.e. opened "r" or "w+" (or even "a+" - ick).

PERLIO_F_ERROR

An error has occurred (for `PerlIO_error()`).

PERLIO_F_TRUNCATE

Truncate file suggested by open mode.

PERLIO_F_APPEND

All writes should be appends.

PERLIO_F_CRLF

Layer is performing Win32-like "\n" mapped to CR,LF for output and CR,LF mapped to "\n" for input. Normally the provided "crlf" layer is the only layer that need bother about this. `PerlIO_binmode()` will mess with this flag rather than add/remove layers if the `PERLIO_K_CANCRLF` bit is set for the layers class.

PERLIO_F_UTF8

Data written to this layer should be UTF-8 encoded; data provided by this layer should be considered UTF-8 encoded. Can be set on any layer by ":utf8" dummy layer. Also set on ":encoding" layer.

PERLIO_F_UNBUF

Layer is unbuffered - i.e. write to next layer down should occur for each write to this layer.

PERLIO_F_WRBUF

The buffer for this layer currently holds data written to it but not sent to next layer.

PERLIO_F_RDBUF

The buffer for this layer currently holds unconsumed data read from layer below.

PERLIO_F_LINEBUF

Layer is line buffered. Write data should be passed to next layer down whenever a "\n" is seen. Any data beyond the "\n" should then be processed.

PERLIO_F_TEMP

File has been `unlink()`ed, or should be deleted on `close()`.

PERLIO_F_OPEN

Handle is open.

PERLIO_F_FASTGETS

This instance of this layer supports the "fast `gets`" interface. Normally set based on `PERLIO_K_FASTGETS` for the class and by the existence of the function(s) in the table. However a class that normally provides that interface may need to avoid it on a particular instance. The "pending" layer needs to do this when it is pushed above a layer which does not support the interface. (Perl's `sv_gets()` does not expect the streams fast `gets` behaviour to change during one "get".)

Methods in Detail

fsize

```
Size_t fsize;
```

Size of the function table. This is compared against the value PerlIO code "knows" as a compatibility check. Future versions *may* be able to tolerate layers compiled against an old version of the headers.

name

```
char * name;
```

The name of the layer whose open() method Perl should invoke on open(). For example if the layer is called APR, you will call:

```
open $fh, ">:APR", ...
```

and Perl knows that it has to invoke the PerlIOAPR_open() method implemented by the APR layer.

size

```
Size_t size;
```

The size of the per-instance data structure, e.g.:

```
sizeof(PerlIOAPR)
```

If this field is zero then `PerlIO_pushed` does not malloc anything and assumes layer's Pushed function will do any required layer stack manipulation - used to avoid malloc/free overhead for dummy layers. If the field is non-zero it must be at least the size of `PerlIOl`, `PerlIO_pushed` will allocate memory for the layer's data structures and link new layer onto the stream's stack. (If the layer's Pushed method returns an error indication the layer is popped again.)

kind

```
IV kind;
```

* PERLIO_K_BUFFERED

  The layer is buffered.

* PERLIO_K_RAW

  The layer is acceptable to have in a binmode(FH) stack - i.e. it does not (or will configure itself not to) transform bytes passing through it.

* PERLIO_K_CANCRLF

  Layer can translate between "\n" and CRLF line ends.

* PERLIO_K_FASTGETS

  Layer allows buffer snooping.

* PERLIO_K_MULTIARG

  Used when the layer's open() accepts more arguments than usual. The extra arguments should come not before the `MODE` argument. When this flag is used it's up to the layer to validate the args.

Pushed

```
IV (*Pushed)(pTHX_ PerlIO *f,const char *mode, SV *arg);
```

The only absolutely mandatory method. Called when the layer is pushed onto the stack. The `mode` argument may be NULL if this occurs post-open. The `arg` will be non-`NULL` if an argument string was passed. In most cases this should call `PerlIOBase_pushed()` to convert `mode` into the appropriate `PERLIO_F_XXXXX` flags in addition to any actions the layer itself takes. If a layer is not expecting an argument it need neither save the one passed to it, nor provide `Getarg()` (it could perhaps `Perl_warn` that the argument was un-expected).

Returns 0 on success. On failure returns -1 and should set errno.

Popped

```
IV (*Popped)(pTHX_ PerlIO *f);
```

Called when the layer is popped from the stack. A layer will normally be popped after `Close()` is called. But a layer can be popped without being closed if the program is dynamically managing layers on the stream. In such cases `Popped()` should free any resources (buffers, translation tables, ...) not held directly in the layer's struct. It should also `Unread()` any unconsumed data that has been read and buffered from the layer below back to that layer, so that it can be re-provided to what ever is now above.

Returns 0 on success and failure. If `Popped()` returns *true* then *perlio.c* assumes that either the layer has popped itself, or the layer is super special and needs to be retained for other reasons. In most cases it should return *false*.

Open

```
PerlIO * (*Open)(...);
```

The `Open()` method has lots of arguments because it combines the functions of perl's `open`, `PerlIO_open`, perl's `sysopen`, `PerlIO_fdopen` and `PerlIO_reopen`. The full prototype is as follows:

```
PerlIO * (*Open)(pTHX_ PerlIO_funcs *tab,
  PerlIO_list_t *layers, IV n,
  const char *mode,
  int fd, int imode, int perm,
  PerlIO *old,
  int narg, SV **args);
```

Open should (perhaps indirectly) call `PerlIO_allocate()` to allocate a slot in the table and associate it with the layers information for the opened file, by calling `PerlIO_push`. The *layers* is an array of all the layers destined for the `PerlIO *`, and any arguments passed to them, *n* is the index into that array of the layer being called. The macro `PerlIOArg` will return a (possibly `NULL`) SV * for the argument passed to the layer.

The *mode* string is an "`fopen()`-like" string which would match the regular expression `/^[I#]?[rwa]\+?[bt]?$/`.

The `'I'` prefix is used during creation of `stdin`..`stderr` via special `PerlIO_fdopen` calls; the `'#'` prefix means that this is `sysopen` and that *imode* and *perm* should be passed to `PerlLIO_open3`; `'r'` means **r**ead, `'w'` means **w**rite and `'a'` means **a**ppend. The `'+'` suffix means that both reading and writing/appending are permitted. The `'b'` suffix means file should be binary, and `'t'` means it is text. (Almost all layers should do the IO in binary mode, and ignore the b/t bits. The `:crlf` layer should be pushed to handle the distinction.)

If *old* is not `NULL` then this is a `PerlIO_reopen`. Perl itself does not use this (yet?) and semantics are a little vague.

If *fd* not negative then it is the numeric file descriptor *fd*, which will be open in a manner compatible with the supplied mode string, the call is thus equivalent to `PerlIO_fdopen`. In this case *nargs* will be zero.

If *nargs* is greater than zero then it gives the number of arguments passed to `open`, otherwise it will be 1 if for example `PerlIO_open` was called. In simple cases SvPV_nolen(*args) is the pathname to open.

If a layer provides `Open()` it should normally call the `Open()` method of next layer down (if any) and then push itself on top if that succeeds. `PerlIOBase_open` is provided to do exactly that, so in most cases you don't have to write your own `Open()` method. If this method is not defined, other layers may have difficulty pushing themselves on top of it during open.

If `PerlIO_push` was performed and open has failed, it must `PerlIO_pop` itself, since if it's not, the layer won't be removed and may cause bad problems.

Returns `NULL` on failure.

Binmode

```
IV        (*Binmode)(pTHX_ PerlIO *f);
```

Optional. Used when `:raw` layer is pushed (explicitly or as a result of binmode(FH)). If not present layer will be popped. If present should configure layer as binary (or pop itself) and return 0. If it returns -1 for error `binmode` will fail with layer still on the stack.

Getarg

```
SV *      (*Getarg)(pTHX_ PerlIO *f,
      CLONE_PARAMS *param, int flags);
```

Optional. If present should return an SV * representing the string argument passed to the layer when it was pushed. e.g. ":encoding(ascii)" would return an SvPV with value "ascii". (*param* and *flags* arguments can be ignored in most cases)

`Dup` uses `Getarg` to retrieve the argument originally passed to `Pushed`, so you must implement this function if your layer has an extra argument to `Pushed` and will ever be `Duped`.

Fileno

```
IV        (*Fileno)(pTHX_ PerlIO *f);
```

Returns the Unix/Posix numeric file descriptor for the handle. Normally `PerlIOBase_fileno()` (which just asks next layer down) will suffice for this.

Returns -1 on error, which is considered to include the case where the layer cannot provide such a file descriptor.

Dup

```
PerlIO * (*Dup)(pTHX_ PerlIO *f, PerlIO *o,
  CLONE_PARAMS *param, int flags);
```

XXX: Needs more docs.

Used as part of the "clone" process when a thread is spawned (in which case param will be non-NULL) and when a stream is being duplicated via '&' in the `open`.

Similar to `Open`, returns PerlIO* on success, `NULL` on failure.

Read

```
SSize_t (*Read)(pTHX_ PerlIO *f, void *vbuf, Size_t count);
```

Basic read operation.

Typically will call `Fill` and manipulate pointers (possibly via the API). `PerlIOBuf_read()` may be suitable for derived classes which provide "fast gets" methods.

Returns actual bytes read, or -1 on an error.

Unread

```
SSize_t (*Unread)(pTHX_ PerlIO *f,
    const void *vbuf, Size_t count);
```

A superset of stdio's `ungetc()`. Should arrange for future reads to see the bytes in `vbuf`. If there is no obviously better implementation then `PerlIOBase_unread()` provides the function by pushing a "fake" "pending" layer above the calling layer.

Returns the number of unread chars.

Write

```
SSize_t (*Write)(PerlIO *f, const void *vbuf, Size_t count);
```

Basic write operation.

Returns bytes written or -1 on an error.

Seek

```
IV (*Seek)(pTHX_ PerlIO *f, Off_t offset, int whence);
```

Position the file pointer. Should normally call its own `Flush` method and then the `Seek` method of next layer down.

Returns 0 on success, -1 on failure.

Tell

```
Off_t (*Tell)(pTHX_ PerlIO *f);
```

Return the file pointer. May be based on layers cached concept of position to avoid overhead.

Returns -1 on failure to get the file pointer.

Close

```
IV (*Close)(pTHX_ PerlIO *f);
```

Close the stream. Should normally call `PerlIOBase_close()` to flush itself and close layers below, and then deallocate any data structures (buffers, translation tables, ...) not held directly in the data structure.

Returns 0 on success, -1 on failure.

Flush

```
IV (*Flush)(pTHX_ PerlIO *f);
```

Should make stream's state consistent with layers below. That is, any buffered write data should be written, and file position of lower layers adjusted for data read from below but not actually consumed. (Should perhaps `Unread()` such data to the lower layer.)

Returns 0 on success, -1 on failure.

Fill

```
IV (*Fill)(pTHX_ PerlIO *f);
```

The buffer for this layer should be filled (for read) from layer below. When you "subclass" PerlIOBuf layer, you want to use its *_read* method and to supply your own fill method, which fills the PerlIOBuf's buffer.

Returns 0 on success, -1 on failure.

Eof

```
IV (*Eof)(pTHX_ PerlIO *f);
```

Return end-of-file indicator. `PerlIOBase_eof()` is normally sufficient.

Returns 0 on end-of-file, 1 if not end-of-file, -1 on error.

Error

```
IV (*Error)(pTHX_ PerlIO *f);
```

Return error indicator. `PerlIOBase_error()` is normally sufficient.

Returns 1 if there is an error (usually when `PERLIO_F_ERROR` is set), 0 otherwise.

Clearerr

```
void (*Clearerr)(pTHX_ PerlIO *f);
```

Clear end-of-file and error indicators. Should call `PerlIOBase_clearerr()` to set the `PERLIO_F_XXXXX` flags, which may suffice.

Setlinebuf

```
void (*Setlinebuf)(pTHX_ PerlIO *f);
```

Mark the stream as line buffered. `PerlIOBase_setlinebuf()` sets the PERLIO_F_LINEBUF flag and is normally sufficient.

Get_base

```
STDCHAR * (*Get_base)(pTHX_ PerlIO *f);
```

Allocate (if not already done so) the read buffer for this layer and return pointer to it. Return NULL on failure.

Get_bufsiz

```
Size_t (*Get_bufsiz)(pTHX_ PerlIO *f);
```

Return the number of bytes that last `Fill()` put in the buffer.

Get_ptr

```
STDCHAR * (*Get_ptr)(pTHX_ PerlIO *f);
```

Return the current read pointer relative to this layer's buffer.

Get_cnt

```
SSize_t (*Get_cnt)(pTHX_ PerlIO *f);
```

Return the number of bytes left to be read in the current buffer.

Set_ptrcnt

```
void (*Set_ptrcnt)(pTHX_ PerlIO *f,
        STDCHAR *ptr, SSize_t cnt);
```

Adjust the read pointer and count of bytes to match `ptr` and/or `cnt`. The application (or layer above) must ensure they are consistent. (Checking is allowed by the paranoid.)

Utilities

To ask for the next layer down use PerlIONext(PerlIO *f).

To check that a PerlIO* is valid use PerlIOValid(PerlIO *f). (All this does is really just to check that the pointer is non-NULL and that the pointer behind that is non-NULL.)

PerlIOBase(PerlIO *f) returns the "Base" pointer, or in other words, the `PerlIOl*` pointer.

PerlIOSelf(PerlIO* f, type) return the PerlIOBase cast to a type.

Perl_PerlIO_or_Base(PerlIO* f, callback, base, failure, args) either calls the *callback* from the functions of the layer *f* (just by the name of the IO function, like "Read") with the *args*, or if there is no such callback, calls the *base* version of the callback with the same args, or if the f is invalid, set errno to EBADF and return *failure*.

Perl_PerlIO_or_fail(PerlIO* f, callback, failure, args) either calls the *callback* of the functions of the layer *f* with the *args*, or if there is no such callback, set errno to EINVAL. Or if the f is invalid, set errno to EBADF and return *failure*.

Perl_PerlIO_or_Base_void(PerlIO* f, callback, base, args) either calls the *callback* of the functions of the layer *f* with the *args*, or if there is no such callback, calls the *base* version of the callback with the

same args, or if the f is invalid, set errno to EBADF.

Perl_PerlIO_or_fail_void(PerlIO* f, callback, args) either calls the *callback* of the functions of the layer *f* with the *args*, or if there is no such callback, set errno to EINVAL. Or if the f is invalid, set errno to EBADF.

Implementing PerlIO Layers

If you find the implementation document unclear or not sufficient, look at the existing PerlIO layer implementations, which include:

* C implementations

  The *perlio.c* and *perliol.h* in the Perl core implement the "unix", "perlio", "stdio", "crlf", "utf8", "byte", "raw", "pending" layers, and also the "mmap" and "win32" layers if applicable. (The "win32" is currently unfinished and unused, to see what is used instead in Win32, see *"Querying the layers of filehandles" in PerlIO* .)

  PerlIO::encoding, PerlIO::scalar, PerlIO::via in the Perl core.

  PerlIO::gzip and APR::PerlIO (mod_perl 2.0) on CPAN.

* Perl implementations

  PerlIO::via::QuotedPrint in the Perl core and PerlIO::via::* on CPAN.

If you are creating a PerlIO layer, you may want to be lazy, in other words, implement only the methods that interest you. The other methods you can either replace with the "blank" methods

```
PerlIOBase_noop_ok
PerlIOBase_noop_fail
```

(which do nothing, and return zero and -1, respectively) or for certain methods you may assume a default behaviour by using a NULL method. The Open method looks for help in the 'parent' layer. The following table summarizes the behaviour:

```
method      behaviour with NULL

Clearerr    PerlIOBase_clearerr
Close       PerlIOBase_close
Dup         PerlIOBase_dup
Eof         PerlIOBase_eof
Error       PerlIOBase_error
Fileno      PerlIOBase_fileno
Fill        FAILURE
Flush       SUCCESS
Getarg      SUCCESS
Get_base    FAILURE
Get_bufsiz  FAILURE
Get_cnt     FAILURE
Get_ptr     FAILURE
Open        INHERITED
Popped      SUCCESS
Pushed      SUCCESS
Read        PerlIOBase_read
Seek        FAILURE
Set_cnt     FAILURE
Set_ptrcnt  FAILURE
Setlinebuf  PerlIOBase_setlinebuf
Tell        FAILURE
Unread      PerlIOBase_unread
```

```
    Write       FAILURE

 FAILURE        Set errno (to EINVAL in Unixish, to LIB$_INVARG in VMS) and
                return -1 (for numeric return values) or NULL (for
pointers)
 INHERITED      Inherited from the layer below
 SUCCESS        Return 0 (for numeric return values) or a pointer
```

Core Layers

The file `perlio.c` provides the following layers:

"unix"

A basic non-buffered layer which calls Unix/POSIX `read()`, `write()`, `lseek()`, `close()`. No buffering. Even on platforms that distinguish between O_TEXT and O_BINARY this layer is always O_BINARY.

"perlio"

A very complete generic buffering layer which provides the whole of PerlIO API. It is also intended to be used as a "base class" for other layers. (For example its `Read()` method is implemented in terms of the `Get_cnt()`/`Get_ptr()`/`Set_ptrcnt()` methods).

"perlio" over "unix" provides a complete replacement for stdio as seen via PerlIO API. This is the default for USE_PERLIO when system's stdio does not permit perl's "fast gets" access, and which do not distinguish between `O_TEXT` and `O_BINARY`.

"stdio"

A layer which provides the PerlIO API via the layer scheme, but implements it by calling system's stdio. This is (currently) the default if system's stdio provides sufficient access to allow perl's "fast gets" access and which do not distinguish between `O_TEXT` and `O_BINARY`.

"crlf"

A layer derived using "perlio" as a base class. It provides Win32-like "\n" to CR,LF translation. Can either be applied above "perlio" or serve as the buffer layer itself. "crlf" over "unix" is the default if system distinguishes between `O_TEXT` and `O_BINARY` opens. (At some point "unix" will be replaced by a "native" Win32 IO layer on that platform, as Win32's read/write layer has various drawbacks.) The "crlf" layer is a reasonable model for a layer which transforms data in some way.

"mmap"

If Configure detects `mmap()` functions this layer is provided (with "perlio" as a "base") which does "read" operations by mmap()ing the file. Performance improvement is marginal on modern systems, so it is mainly there as a proof of concept. It is likely to be unbundled from the core at some point. The "mmap" layer is a reasonable model for a minimalist "derived" layer.

"pending"

An "internal" derivative of "perlio" which can be used to provide Unread() function for layers which have no buffer or cannot be bothered. (Basically this layer's `Fill()` pops itself off the stack and so resumes reading from layer below.)

"raw"

A dummy layer which never exists on the layer stack. Instead when "pushed" it actually pops the stack removing itself, it then calls Binmode function table entry on all the layers in the stack - normally this (via PerlIOBase_binmode) removes any layers which do not have `PERLIO_K_RAW` bit set. Layers can modify that behaviour by defining their own Binmode entry.

"utf8"

Another dummy layer. When pushed it pops itself and sets the `PERLIO_F_UTF8` flag on the layer which was (and now is once more) the top of the stack.

In addition *perlio.c* also provides a number of `PerlIOBase_xxxx()` functions which are intended to be used in the table slots of classes which do not need to do anything special for a particular method.

Extension Layers

Layers can be made available by extension modules. When an unknown layer is encountered the PerlIO code will perform the equivalent of :

```
use PerlIO 'layer';
```

Where *layer* is the unknown layer. *PerlIO.pm* will then attempt to:

```
require PerlIO::layer;
```

If after that process the layer is still not defined then the `open` will fail.

The following extension layers are bundled with perl:

":encoding"

```
use Encoding;
```

makes this layer available, although *PerlIO.pm* "knows" where to find it. It is an example of a layer which takes an argument as it is called thus:

```
open( $fh, "<:encoding(iso-8859-7)", $pathname );
```

":scalar"

Provides support for reading data from and writing data to a scalar.

```
open( $fh, "+<:scalar", \$scalar );
```

When a handle is so opened, then reads get bytes from the string value of *$scalar*, and writes change the value. In both cases the position in *$scalar* starts as zero but can be altered via `seek`, and determined via `tell`.

Please note that this layer is implied when calling open() thus:

```
open( $fh, "+<", \$scalar );
```

":via"

Provided to allow layers to be implemented as Perl code. For instance:

```
use PerlIO::via::StripHTML;
open( my $fh, "<:via(StripHTML)", "index.html" );
```

See *PerlIO::via* for details.

TODO

Things that need to be done to improve this document.

- Explain how to make a valid fh without going through open()(i.e. apply a layer). For example if the file is not opened through perl, but we want to get back a fh, like it was opened by Perl.

  How PerlIO_apply_layera fits in, where its docs, was it made public?

  Currently the example could be something like this:

  ```
  PerlIO *foo_to_PerlIO(pTHX_ char *mode, ...)
  ```

```
{
    char *mode; /* "w", "r", etc */
    const char *layers = ":APR"; /* the layer name */
    PerlIO *f = PerlIO_allocate(aTHX);
    if (!f) {
        return NULL;
    }

    PerlIO_apply_layers(aTHX_ f, mode, layers);

    if (f) {
        PerlIOAPR *st = PerlIOSelf(f, PerlIOAPR);
        /* fill in the st struct, as in _open() */
        st->file = file;
        PerlIOBase(f)->flags |= PERLIO_F_OPEN;

        return f;
    }
    return NULL;
}
```

- fix/add the documentation in places marked as XXX.
- The handling of errors by the layer is not specified. e.g. when $! should be set explicitly, when the error handling should be just delegated to the top layer.

  Probably give some hints on using SETERRNO() or pointers to where they can be found.
- I think it would help to give some concrete examples to make it easier to understand the API. Of course I agree that the API has to be concise, but since there is no second document that is more of a guide, I think that it'd make it easier to start with the doc which is an API, but has examples in it in places where things are unclear, to a person who is not a PerlIO guru (yet).