NAME

perlapio - perl's IO abstraction interface.

SYNOPSIS

```
    #define PERLIO_NOT_STDIO 0    /* For co-existence with stdio only */
    #include <perlio.h>           /* Usually via #include <perl.h> */

    PerlIO *PerlIO_stdin(void);
    PerlIO *PerlIO_stdout(void);
    PerlIO *PerlIO_stderr(void);

    PerlIO *PerlIO_open(const char *path,const char *mode);
    PerlIO *PerlIO_fdopen(int fd, const char *mode);
    PerlIO *PerlIO_reopen(const char *path, const char *mode, PerlIO *old);
  /* deprecated */
    int     PerlIO_close(PerlIO *f);

    int     PerlIO_stdoutf(const char *fmt,...)
    int     PerlIO_puts(PerlIO *f,const char *string);
    int     PerlIO_putc(PerlIO *f,int ch);
    int     PerlIO_write(PerlIO *f,const void *buf,size_t numbytes);
    int     PerlIO_printf(PerlIO *f, const char *fmt,...);
    int     PerlIO_vprintf(PerlIO *f, const char *fmt, va_list args);
    int     PerlIO_flush(PerlIO *f);

    int     PerlIO_eof(PerlIO *f);
    int     PerlIO_error(PerlIO *f);
    void    PerlIO_clearerr(PerlIO *f);

    int     PerlIO_getc(PerlIO *d);
    int     PerlIO_ungetc(PerlIO *f,int ch);
    int     PerlIO_read(PerlIO *f, void *buf, size_t numbytes);

    int     PerlIO_fileno(PerlIO *f);

    void    PerlIO_setlinebuf(PerlIO *f);

    Off_t   PerlIO_tell(PerlIO *f);
    int     PerlIO_seek(PerlIO *f, Off_t offset, int whence);
    void    PerlIO_rewind(PerlIO *f);

    int     PerlIO_getpos(PerlIO *f, SV *save);        /* prototype changed
  */
    int     PerlIO_setpos(PerlIO *f, SV *saved);       /* prototype changed
  */

    int     PerlIO_fast_gets(PerlIO *f);
    int     PerlIO_has_cntptr(PerlIO *f);
    int     PerlIO_get_cnt(PerlIO *f);
    char   *PerlIO_get_ptr(PerlIO *f);
    void    PerlIO_set_ptrcnt(PerlIO *f, char *ptr, int count);
```

```
    int     PerlIO_canset_cnt(PerlIO *f);             /* deprecated */
    void    PerlIO_set_cnt(PerlIO *f, int count);     /* deprecated */

    int     PerlIO_has_base(PerlIO *f);
    char   *PerlIO_get_base(PerlIO *f);
    int     PerlIO_get_bufsiz(PerlIO *f);

    PerlIO *PerlIO_importFILE(FILE *stdio, const char *mode);
    FILE   *PerlIO_exportFILE(PerlIO *f, int flags);
    FILE   *PerlIO_findFILE(PerlIO *f);
    void    PerlIO_releaseFILE(PerlIO *f,FILE *stdio);

    int     PerlIO_apply_layers(PerlIO *f, const char *mode, const char
*layers);
    int     PerlIO_binmode(PerlIO *f, int ptype, int imode, const char
*layers);
    void    PerlIO_debug(const char *fmt,...)
```

DESCRIPTION

Perl's source code, and extensions that want maximum portability, should use the above functions instead of those defined in ANSI C's *stdio.h*. The perl headers (in particular "perlio.h") will `#define` them to the I/O mechanism selected at Configure time.

The functions are modeled on those in *stdio.h*, but parameter order has been "tidied up a little".

`PerlIO *` takes the place of FILE *. Like FILE * it should be treated as opaque (it is probably safe to assume it is a pointer to something).

There are currently three implementations:

1. USE_STDIO

   All above are #define'd to stdio functions or are trivial wrapper functions which call stdio. In this case *only* PerlIO * is a FILE *. This has been the default implementation since the abstraction was introduced in perl5.003_02.

2. USE_SFIO

   A "legacy" implementation in terms of the "sfio" library. Used for some specialist applications on Unix machines ("sfio" is not widely ported away from Unix). Most of above are #define'd to the sfio functions. PerlIO * is in this case Sfio_t *.

3. USE_PERLIO

   Introduced just after perl5.7.0, this is a re-implementation of the above abstraction which allows perl more control over how IO is done as it decouples IO from the way the operating system and C library choose to do things. For USE_PERLIO PerlIO * has an extra layer of indirection - it is a pointer-to-a-pointer. This allows the PerlIO * to remain with a known value while swapping the implementation around underneath *at run time*. In this case all the above are true (but very simple) functions which call the underlying implementation.

   This is the only implementation for which `PerlIO_apply_layers()` does anything "interesting".

   The USE_PERLIO implementation is described in *perliol*.

Because "perlio.h" is a thin layer (for efficiency) the semantics of these functions are somewhat dependent on the underlying implementation. Where these variations are understood they are noted below.

Unless otherwise noted, functions return 0 on success, or a negative value (usually EOF which is usually -1) and set errno on error.

PerlIO_stdin(), **PerlIO_stdout()**, **PerlIO_stderr()**

Use these rather than stdin, stdout, stderr. They are written to look like "function calls" rather than variables because this makes it easier to *make them* function calls if platform cannot export data to loaded modules, or if (say) different "threads" might have different values.

PerlIO_open(path, mode), **PerlIO_fdopen(fd,mode)**

These correspond to fopen()/fdopen() and the arguments are the same. Return NULL and set errno if there is an error. There may be an implementation limit on the number of open handles, which may be lower than the limit on the number of open files - errno may not be set when NULL is returned if this limit is exceeded.

PerlIO_reopen(path,mode,f)

While this currently exists in all three implementations perl itself does not use it. *As perl does not use it, it is not well tested.*

Perl prefers to dup the new low-level descriptor to the descriptor used by the existing PerlIO. This may become the behaviour of this function in the future.

PerlIO_printf(f,fmt,...), **PerlIO_vprintf(f,fmt,a)**

These are fprintf()/vfprintf() equivalents.

PerlIO_stdoutf(fmt,...)

This is printf() equivalent. printf is #defined to this function, so it is (currently) legal to use printf(fmt,...) in perl sources.

PerlIO_read(f,buf,count), **PerlIO_write(f,buf,count)**

These correspond functionally to fread() and fwrite() but the arguments and return values are different. The PerlIO_read() and PerlIO_write() signatures have been modeled on the more sane low level read() and write() functions instead: The "file" argument is passed first, there is only one "count", and the return value can distinguish between error and EOF.

Returns a byte count if successful (which may be zero or positive), returns negative value and sets errno on error. Depending on implementation errno may be EINTR if operation was interrupted by a signal.

PerlIO_close(f)

Depending on implementation errno may be EINTR if operation was interrupted by a signal.

PerlIO_puts(f,s), **PerlIO_putc(f,c)**

These correspond to fputs() and fputc(). Note that arguments have been revised to have "file" first.

PerlIO_ungetc(f,c)

This corresponds to ungetc(). Note that arguments have been revised to have "file" first. Arranges that next read operation will return the byte **c**. Despite the implied "character" in the name only values in the range 0..0xFF are defined. Returns the byte **c** on success or -1 (EOF) on error. The number of bytes that can be "pushed back" may vary, only 1 character is certain, and then only if it is the last character that was read from the handle.

PerlIO_getc(f)

This corresponds to getc(). Despite the c in the name only byte range 0..0xFF is supported. Returns the character read or -1 (EOF) on error.

PerlIO_eof(f)

This corresponds to feof(). Returns a true/false indication of whether the handle is at end of file. For terminal devices this may or may not be "sticky" depending on the implementation. The flag is cleared by PerlIO_seek(), or PerlIO_rewind().

PerlIO_error(f)

This corresponds to ferror(). Returns a true/false indication of whether there has been an IO error on the handle.

PerlIO_fileno(f)

This corresponds to fileno(), note that on some platforms, the meaning of "fileno" may not match Unix. Returns -1 if the handle has no open descriptor associated with it.

PerlIO_clearerr(f)

This corresponds to clearerr(), i.e., clears 'error' and (usually) 'eof' flags for the "stream". Does not return a value.

PerlIO_flush(f)

This corresponds to fflush(). Sends any buffered write data to the underlying file. If called with `NULL` this may flush all open streams (or core dump with some USE_STDIO implementations). Calling on a handle open for read only, or on which last operation was a read of some kind may lead to undefined behaviour on some USE_STDIO implementations. The USE_PERLIO (layers) implementation tries to behave better: it flushes all open streams when passed `NULL`, and attempts to retain data on read streams either in the buffer or by seeking the handle to the current logical position.

PerlIO_seek(f,offset,whence)

This corresponds to fseek(). Sends buffered write data to the underlying file, or discards any buffered read data, then positions the file descriptor as specified by **offset** and **whence** (sic). This is the correct thing to do when switching between read and write on the same handle (see issues with PerlIO_flush() above). Offset is of type `Off_t` which is a perl Configure value which may not be same as stdio's `off_t`.

PerlIO_tell(f)

This corresponds to ftell(). Returns the current file position, or (Off_t) -1 on error. May just return value system "knows" without making a system call or checking the underlying file descriptor (so use on shared file descriptors is not safe without a PerlIO_seek()). Return value is of type `Off_t` which is a perl Configure value which may not be same as stdio's `off_t`.

PerlIO_getpos(f,p), **PerlIO_setpos(f,p)**

These correspond (loosely) to fgetpos() and fsetpos(). Rather than stdio's Fpos_t they expect a "Perl Scalar Value" to be passed. What is stored there should be considered opaque. The layout of the data may vary from handle to handle. When not using stdio or if platform does not have the stdio calls then they are implemented in terms of PerlIO_tell() and PerlIO_seek().

PerlIO_rewind(f)

This corresponds to rewind(). It is usually defined as being

```
PerlIO_seek(f,(Off_t)0L, SEEK_SET);
PerlIO_clearerr(f);
```

PerlIO_tmpfile()

This corresponds to tmpfile(), i.e., returns an anonymous PerlIO or NULL on error. The system will attempt to automatically delete the file when closed. On Unix the file is usually `unlink`-ed just after it is created so it does not matter how it gets closed. On other systems the file may only be deleted if closed via PerlIO_close() and/or the program exits via `exit`. Depending on the implementation there may be "race conditions" which allow other processes access to the

file, though in general it will be safer in this regard than ad. hoc. schemes.

PerlIO_setlinebuf(f)

This corresponds to setlinebuf(). Does not return a value. What constitutes a "line" is implementation dependent but usually means that writing "\n" flushes the buffer. What happens with things like "this\nthat" is uncertain. (Perl core uses it *only* when "dumping"; it has nothing to do with $| auto-flush.)

Co-existence with stdio

There is outline support for co-existence of PerlIO with stdio. Obviously if PerlIO is implemented in terms of stdio there is no problem. However in other cases then mechanisms must exist to create a FILE * which can be passed to library code which is going to use stdio calls.

The first step is to add this line:

```
#define PERLIO_NOT_STDIO 0
```

before including any perl header files. (This will probably become the default at some point). That prevents "perlio.h" from attempting to #define stdio functions onto PerlIO functions.

XS code is probably better using "typemap" if it expects FILE * arguments. The standard typemap will be adjusted to comprehend any changes in this area.

PerlIO_importFILE(f,mode)

Used to get a PerlIO * from a FILE *.

The mode argument should be a string as would be passed to fopen/PerlIO_open. If it is NULL then - for legacy support - the code will (depending upon the platform and the implementation) either attempt to empirically determine the mode in which *f* is open, or use "r+" to indicate a read/write stream.

Once called the FILE * should *ONLY* be closed by calling `PerlIO_close()` on the returned PerlIO *.

The PerlIO is set to textmode. Use PerlIO_binmode if this is not the desired mode.

This is **not** the reverse of PerlIO_exportFILE().

PerlIO_exportFILE(f,mode)

Given a PerlIO * create a 'native' FILE * suitable for passing to code expecting to be compiled and linked with ANSI C *stdio.h*. The mode argument should be a string as would be passed to fopen/PerlIO_open. If it is NULL then - for legacy support - the FILE * is opened in same mode as the PerlIO *.

The fact that such a FILE * has been 'exported' is recorded, (normally by pushing a new :stdio "layer" onto the PerlIO *), which may affect future PerlIO operations on the original PerlIO *. You should not call `fclose()` on the file unless you call `PerlIO_releaseFILE()` to disassociate it from the PerlIO *. (Do not use PerlIO_importFILE() for doing the disassociation.)

Calling this function repeatedly will create a FILE * on each call (and will push an :stdio layer each time as well).

PerlIO_releaseFILE(p,f)

Calling PerlIO_releaseFILE informs PerlIO that all use of FILE * is complete. It is removed from the list of 'exported' FILE *s, and the associated PerlIO * should revert to its original behaviour.

Use this to disassociate a file from a PerlIO * that was associated using PerlIO_exportFILE().

PerlIO_findFILE(f)

Returns a native FILE * used by a stdio layer. If there is none, it will create one with PerlIO_exportFILE. In either case the FILE * should be considered as belonging to PerlIO subsystem and should only be closed by calling `PerlIO_close()`.

"Fast gets" Functions

In addition to standard-like API defined so far above there is an "implementation" interface which allows perl to get at internals of PerlIO. The following calls correspond to the various FILE_xxx macros determined by Configure - or their equivalent in other implementations. This section is really of interest to only those concerned with detailed perl-core behaviour, implementing a PerlIO mapping or writing code which can make use of the "read ahead" that has been done by the IO system in the same way perl does. Note that any code that uses these interfaces must be prepared to do things the traditional way if a handle does not support them.

PerlIO_fast_gets(f)

Returns true if implementation has all the interfaces required to allow perl's `sv_gets` to "bypass" normal IO mechanism. This can vary from handle to handle.

```
PerlIO_fast_gets(f) = PerlIO_has_cntptr(f) && \
                      PerlIO_canset_cnt(f) && \
                      `Can set pointer into buffer'
```

PerlIO_has_cntptr(f)

Implementation can return pointer to current position in the "buffer" and a count of bytes available in the buffer. Do not use this - use PerlIO_fast_gets.

PerlIO_get_cnt(f)

Return count of readable bytes in the buffer. Zero or negative return means no more bytes available.

PerlIO_get_ptr(f)

Return pointer to next readable byte in buffer, accessing via the pointer (dereferencing) is only safe if PerlIO_get_cnt() has returned a positive value. Only positive offsets up to value returned by PerlIO_get_cnt() are allowed.

PerlIO_set_ptrcnt(f,p,c)

Set pointer into buffer, and a count of bytes still in the buffer. Should be used only to set pointer to within range implied by previous calls to `PerlIO_get_ptr` and `PerlIO_get_cnt`. The two values *must* be consistent with each other (implementation may only use one or the other or may require both).

PerlIO_canset_cnt(f)

Implementation can adjust its idea of number of bytes in the buffer. Do not use this - use PerlIO_fast_gets.

PerlIO_set_cnt(f,c)

Obscure - set count of bytes in the buffer. Deprecated. Only usable if PerlIO_canset_cnt() returns true. Currently used in only doio.c to force count less than -1 to -1. Perhaps should be PerlIO_set_empty or similar. This call may actually do nothing if "count" is deduced from pointer and a "limit". Do not use this - use PerlIO_set_ptrcnt().

PerlIO_has_base(f)

Returns true if implementation has a buffer, and can return pointer to whole buffer and its size. Used by perl for **-T** / **-B** tests. Other uses would be very obscure...

PerlIO_get_base(f)

Return *start* of buffer. Access only positive offsets in the buffer up to the value returned by

PerlIO_get_bufsiz(f) Deprecated synonym for PerlIO_bufsiz().

Return the *total number of bytes* in the buffer, this is neither the number that can be read, nor the amount of memory allocated to the buffer. Rather it is what the operating system and/or implementation happened to `read()` (or whatever) last time IO was requested.

Other Functions

PerlIO_apply_layers(f,mode,layers)

The new interface to the USE_PERLIO implementation. The layers ":crlf" and ":raw" are only ones allowed for other implementations and those are silently ignored. (As of perl5.8 ":raw" is deprecated.) Use PerlIO_binmode() below for the portable case.

PerlIO_binmode(f,ptype,imode,layers)

The hook used by perl's `binmode` operator. **ptype** is perl's character for the kind of IO:

'<' read

'>' write

'+' read/write

imode is `O_BINARY` or `O_TEXT`.

layers is a string of layers to apply, only ":crlf" makes sense in the non USE_PERLIO case. (As of perl5.8 ":raw" is deprecated in favour of passing NULL.)

Portable cases are:

```
    PerlIO_binmode(f,ptype,O_BINARY,NULL);
and
    PerlIO_binmode(f,ptype,O_TEXT,":crlf");
```

On Unix these calls probably have no effect whatsoever. Elsewhere they alter "\n" to CR,LF translation and possibly cause a special text "end of file" indicator to be written or honoured on read. The effect of making the call after doing any IO to the handle depends on the implementation. (It may be ignored, affect any data which is already buffered as well, or only apply to subsequent data.)

PerlIO_debug(fmt,...)

PerlIO_debug is a printf()-like function which can be used for debugging. No return value. Its main use is inside PerlIO where using real printf, warn() etc. would recursively call PerlIO and be a problem.

PerlIO_debug writes to the file named by $ENV{'PERLIO_DEBUG'} typical use might be

```
  Bourne shells (sh, ksh, bash, zsh, ash, ...):
   PERLIO_DEBUG=/dev/tty ./perl somescript some args

  Csh/Tcsh:
   setenv PERLIO_DEBUG /dev/tty
   ./perl somescript some args

  If you have the "env" utility:
   env PERLIO_DEBUG=/dev/tty ./perl somescript some args

  Win32:
   set PERLIO_DEBUG=CON
   perl somescript some args
```

If $ENV{'PERLIO_DEBUG'} is not set PerlIO_debug() is a no-op.