NAME

IO::Handle - supply object methods for I/O handles

SYNOPSIS

```
use IO::Handle;

$io = IO::Handle->new();
if ($io->fdopen(fileno(STDIN),"r")) {
    print $io->getline;
    $io->close;
}

$io = IO::Handle->new();
if ($io->fdopen(fileno(STDOUT),"w")) {
    $io->print("Some text\n");
}

# setvbuf is not available by default on Perls 5.8.0 and later.
use IO::Handle '_IOLBF';
$io->setvbuf($buffer_var, _IOLBF, 1024);

undef $io;       # automatically closes the file if it's open

autoflush STDOUT 1;
```

DESCRIPTION

`IO::Handle` is the base class for all other IO handle classes. It is not intended that objects of `IO::Handle` would be created directly, but instead `IO::Handle` is inherited from by several other classes in the IO hierarchy.

If you are reading this documentation, looking for a replacement for the `FileHandle` package, then I suggest you read the documentation for `IO::File` too.

CONSTRUCTOR

new ()

Creates a new `IO::Handle` object.

new_from_fd (FD, MODE)

Creates an `IO::Handle` like `new` does. It requires two parameters, which are passed to the method `fdopen`; if the fdopen fails, the object is destroyed. Otherwise, it is returned to the caller.

METHODS

See *perlfunc* for complete descriptions of each of the following supported `IO::Handle` methods, which are just front ends for the corresponding built-in functions:

```
$io->close
$io->eof
$io->fcntl( FUNCTION, SCALAR )
$io->fileno
$io->format_write( [FORMAT_NAME] )
$io->getc
$io->ioctl( FUNCTION, SCALAR )
$io->read ( BUF, LEN, [OFFSET] )
```

```
$io->print ( ARGS )
$io->printf ( FMT, [ARGS] )
$io->say ( ARGS )
$io->stat
$io->sysread ( BUF, LEN, [OFFSET] )
$io->syswrite ( BUF, [LEN, [OFFSET]] )
$io->truncate ( LEN )
```

See *perlvar* for complete descriptions of each of the following supported `IO::Handle` methods. All of them return the previous value of the attribute and takes an optional single argument that when given will set the value. If no argument is given the previous value is unchanged (except for $io->autoflush will actually turn ON autoflush by default).

```
$io->autoflush ( [BOOL] )                       $|
$io->format_page_number( [NUM] )                $%
$io->format_lines_per_page( [NUM] )             $=
$io->format_lines_left( [NUM] )                 $-
$io->format_name( [STR] )                       $~
$io->format_top_name( [STR] )                   $^
$io->input_line_number( [NUM])                  $.
```

The following methods are not supported on a per-filehandle basis.

```
IO::Handle->format_line_break_characters( [STR] ) $:
IO::Handle->format_formfeed( [STR])               $^L
IO::Handle->output_field_separator( [STR] )       $,
IO::Handle->output_record_separator( [STR] )      $\

IO::Handle->input_record_separator( [STR] )       $/
```

Furthermore, for doing normal I/O you might need these:

$io->fdopen (FD, MODE)

> `fdopen` is like an ordinary `open` except that its first parameter is not a filename but rather a file handle name, an IO::Handle object, or a file descriptor number. (For the documentation of the `open` method, see *IO::File*.)

$io->opened

> Returns true if the object is currently a valid file descriptor, false otherwise.

$io->getline

> This works like <$io> described in *"I/O Operators" in perlop* except that it's more readable and can be safely called in a list context but still returns just one line. If used as the conditional within a `while` or C-style `for` loop, however, you will need to emulate the functionality of <$io> with `defined($_ = $io->getline)`.

$io->getlines

> This works like <$io> when called in a list context to read all the remaining lines in a file, except that it's more readable. It will also croak() if accidentally called in a scalar context.

$io->ungetc (ORD)

> Pushes a character with the given ordinal value back onto the given handle's input stream. Only one character of pushback per handle is guaranteed.

$io->write (BUF, LEN [, OFFSET])

This `write` is somewhat like `write` found in C, in that it is the opposite of read. The wrapper for the perl `write` function is called `format_write`. However, whilst the C `write` function returns the number of bytes written, this `write` function simply returns true if successful (like `print`). A more C-like `write` is `syswrite` (see above).

$io->error

Returns a true value if the given handle has experienced any errors since it was opened or since the last call to `clearerr`, or if the handle is invalid. It only returns false for a valid handle with no outstanding errors.

$io->clearerr

Clear the given handle's error indicator. Returns -1 if the handle is invalid, 0 otherwise.

$io->sync

`sync` synchronizes a file's in-memory state with that on the physical medium. `sync` does not operate at the perlio api level, but operates on the file descriptor (similar to sysread, sysseek and systell). This means that any data held at the perlio api level will not be synchronized. To synchronize data that is buffered at the perlio api level you must use the flush method. `sync` is not implemented on all platforms. Returns "0 but true" on success, `undef` on error, `undef` for an invalid handle. See *fsync(3c)*.

$io->flush

`flush` causes perl to flush any buffered data at the perlio api level. Any unread data in the buffer will be discarded, and any unwritten data will be written to the underlying file descriptor. Returns "0 but true" on success, `undef` on error.

$io->printflush (ARGS)

Turns on autoflush, print ARGS and then restores the autoflush status of the `IO::Handle` object. Returns the return value from print.

$io->blocking ([BOOL])

If called with an argument `blocking` will turn on non-blocking IO if `BOOL` is false, and turn it off if `BOOL` is true.

`blocking` will return the value of the previous setting, or the current setting if `BOOL` is not given.

If an error occurs `blocking` will return undef and `$!` will be set.

If the C functions setbuf() and/or setvbuf() are available, then `IO::Handle::setbuf` and `IO::Handle::setvbuf` set the buffering policy for an IO::Handle. The calling sequences for the Perl functions are the same as their C counterparts--including the constants `_IOFBF`, `_IOLBF`, and `_IONBF` for setvbuf()--except that the buffer parameter specifies a scalar variable to use as a buffer. You should only change the buffer before any I/O, or immediately after calling flush.

WARNING: The IO::Handle::setvbuf() is not available by default on Perls 5.8.0 and later because setvbuf() is rather specific to using the stdio library, while Perl prefers the new perlio subsystem instead.

WARNING: A variable used as a buffer by `setbuf` or `setvbuf` **must not be modified** in any way until the IO::Handle is closed or `setbuf` or `setvbuf` is called again, or memory corruption may result! Remember that the order of global destruction is undefined, so even if your buffer variable remains in scope until program termination, it may be undefined before the file IO::Handle is closed. Note that you need to import the constants `_IOFBF`, `_IOLBF`, and `_IONBF` explicitly. Like C, setbuf returns nothing. setvbuf returns "0 but true", on success, `undef` on failure.

Lastly, there is a special method for working under **-T** and setuid/gid scripts:

$io->untaint

Marks the object as taint-clean, and as such data read from it will also be considered taint-clean. Note that this is a very trusting action to take, and appropriate consideration for the data source and potential vulnerability should be kept in mind. Returns 0 on success, -1 if setting the taint-clean flag failed. (eg invalid handle)

NOTE

An `IO::Handle` object is a reference to a symbol/GLOB reference (see the `Symbol` package). Some modules that inherit from `IO::Handle` may want to keep object related variables in the hash table part of the GLOB. In an attempt to prevent modules trampling on each other I propose the that any such module should prefix its variables with its own name separated by _'s. For example the IO::Socket module keeps a `timeout` variable in 'io_socket_timeout'.

SEE ALSO

perlfunc, *"I/O Operators" in perlop*, *IO::File*

BUGS

Due to backwards compatibility, all filehandles resemble objects of class `IO::Handle`, or actually classes derived from that class. They actually aren't. Which means you can't derive your own class from `IO::Handle` and inherit those methods.

HISTORY

Derived from FileHandle.pm by Graham Barr <*gbarr@pobox.com*>