Table of Contents

Parent

Object

Included Modules

File::File::Constants
Enumerable

Methods

class IO

An instance of class IO (commonly called a *stream*) represents an input/output stream in the underlying operating system. Class IO is the basis for input and output in Ruby.

Class `File` is the only class in the Ruby core that is a subclass of IO. Some classes in the Ruby standard library are also subclasses of IO; these include TCPSocket and UDPSocket.

The global constant `ARGF` (also accessible as `$<`) provides an IO-like stream that allows access to all file paths found in ARGV (or found in STDIN if ARGV is empty). `ARGF` is not itself a subclass of IO.

Class StringIO provides an IO-like stream that handles a `String`. StringIO is not itself a subclass of IO.

Important objects based on IO include:

- $stdin.
- $stdout.
- $stderr.
- Instances of class `File`.

An instance of IO may be created using:

- `IO.new`: returns a new IO object for the given integer file descriptor.
- `IO.open`: passes a new IO object to the given block.
- `IO.popen`: returns a new IO object that is connected to the $stdin and $stdout of a newly-launched subprocess.
- `Kernel#open`: Returns a new IO object connected to a given source: stream, file, or subprocess.

Like a `File` stream, an IO stream has:

- A read/write mode, which may be read-only, write-only, or read/write; see Read/Write Mode.
- A data mode, which may be text-only or binary; see Data Mode.
- Internal and external encodings; see Encodings.

And like other IO streams, it has:

- A position, which determines where in the stream the next read or write is to occur; see Position.
- A line number, which is a special, line-oriented, “position” (different from the position mentioned above); see Line Number.

Extension `io/console`

Extension `io/console` provides numerous methods for interacting with the console; requiring it adds numerous methods to class IO.

Example Files

Many examples here use these variables:

```
# English text with newlines.
text = <<~EOT
```

```
  First line
  Second line

  Fourth line
  Fifth line
EOT

# Russian text.
russian = "\u{442 435 441 442}" # => "тест"

# Binary data.
data = "\u9990\u9991\u9992\u9993\u9994"

# Text file.
File.write('t.txt', text)

# File with Russian text.
File.write('t.rus', russian)

# File with binary data.
f = File.new('t.dat', 'wb:UTF-16')
f.write(data)
f.close
```

Open Options

A number of IO methods accept optional keyword arguments that determine how a new stream is to be opened:

- `:mode` : Stream mode.
- `:flags` : `Integer` file open flags; If `mode` is also given, the two are bitwise-ORed.
- `:external_encoding` : External encoding for the stream.
- `:internal_encoding` : Internal encoding for the stream. `'-'` is a synonym for the default internal encoding. If the value is `nil` no conversion occurs.
- `:encoding` : Specifies external and internal encodings as `'extern:intern'`.
- `:textmode` : If a truthy value, specifies the mode as text-only, binary otherwise.
- `:binmode` : If a truthy value, specifies the mode as binary, text-only otherwise.
- `:autoclose` : If a truthy value, specifies that the `fd` will close when the stream closes; otherwise it remains open.
- `:path:` If a string value is provided, it is used in `inspect` and is available as `path` method.

Also available are the options offered in `String#encode`, which may control conversion between external and internal encoding.

Basic IO

You can perform basic stream IO with these methods, which typically operate on multi-byte strings:

- `IO#read`: Reads and returns some or all of the remaining bytes from the stream.
- `IO#write`: Writes zero or more strings to the stream; each given object that is not already a string is converted via `to_s`.

Position

An IO stream has a nonnegative integer *position*, which is the byte offset at which the next read or write is to occur. A new stream has position zero (and line number zero); method `rewind` resets the position (and line number) to zero.

The relevant methods:

- `IO#tell` (aliased as `#pos`): Returns the current position (in bytes) in the stream.
- `IO#pos=`: Sets the position of the stream to a given integer `new_position` (in bytes).
- `IO#seek`: Sets the position of the stream to a given integer `offset` (in bytes), relative to a given position `whence` (indicating the beginning, end, or current position).
- `IO#rewind`: Positions the stream at the beginning (also resetting the line number).

Open and Closed Streams

A new IO stream may be open for reading, open for writing, or both.

A stream is automatically closed when claimed by the garbage collector.

Attempted reading or writing on a closed stream raises an exception.

The relevant methods:

- `IO#close`: Closes the stream for both reading and writing.
- `IO#close_read`: Closes the stream for reading.
- `IO#close_write`: Closes the stream for writing.
- `IO#closed?`: Returns whether the stream is closed.

End-of-Stream

You can query whether a stream is positioned at its end:

- `IO#eof?` (also aliased as `#eof`): Returns whether the stream is at end-of-stream.

You can reposition to end-of-stream by using method `IO#seek`:

```
f = File.new('t.txt')
f.eof? # => false
f.seek(0, :END)
f.eof? # => true
f.close
```

Or by reading all stream content (which is slower than using `IO#seek`):

```
f.rewind
f.eof? # => false
f.read # => "First line\nSecond line\n\nFourth line\nFifth line\n"
f.eof? # => true
```

Line IO

You can read an IO stream line-by-line using these methods:

- `IO#each_line`: Reads each remaining line, passing it to the given block.
- `IO#gets`: Returns the next line.
- `IO#readline`: Like `gets`, but raises an exception at end-of-stream.
- `IO#readlines`: Returns all remaining lines in an array.

Each of these reader methods accepts:

- An optional line separator, `sep`; see Line Separator.
- An optional line-size limit, `limit`; see Line Limit.

For each of these reader methods, reading may begin mid-line, depending on the stream's position; see Position:

```
f = File.new('t.txt')
f.pos = 27
f.each_line {|line| p line }
f.close
```

Output:

```
"rth line\n"
"Fifth line\n"
```

You can write to an IO stream line-by-line using this method:

- IO#puts : Writes objects to the stream.

Line Separator

Each of these methods uses a *line separator*, which is the string that delimits lines:

- IO.foreach.
- IO.readlines.
- IO#each_line.
- IO#gets.
- IO#readline.
- IO#readlines.

The default line separator is the given by the global variable `$/`, whose value is by default `"\n"`. The line to be read next is all data from the current position to the next line separator:

```
f = File.new('t.txt')
f.gets # => "First line\n"
f.gets # => "Second line\n"
f.gets # => "\n"
f.gets # => "Fourth line\n"
f.gets # => "Fifth line\n"
f.close
```

You can specify a different line separator:

```
f = File.new('t.txt')
f.gets('l')   # => "First l"
f.gets('li')  # => "ine\nSecond li"
f.gets('lin') # => "ne\n\nFourth lin"
f.gets        # => "e\n"
f.close
```

There are two special line separators:

- `nil`: The entire stream is read into a single string:

```
f = File.new('t.txt')
f.gets(nil) # => "First line\nSecond line\n\nFourth line\nFifth line\
f.close
```

- `''` (the empty string): The next “paragraph” is read (paragraphs being separated by two consecutive line separators):

```
f = File.new('t.txt')
f.gets('') # => "First line\nSecond line\n\n"
```

```
f.gets('') # => "Fourth line\nFifth line\n"
f.close
```

Line Limit

Each of these methods uses a *line limit*, which specifies that the number of bytes returned may not be (much) longer than the given `limit`;

- IO.foreach.
- IO.readlines.
- IO#each_line.
- IO#gets.
- IO#readline.
- IO#readlines.

A multi-byte character will not be split, and so a line may be slightly longer than the given limit.

If `limit` is not given, the line is determined only by `sep`.

```
# Text with 1-byte characters.
File.open('t.txt') {|f| f.gets(1) }  # => "F"
File.open('t.txt') {|f| f.gets(2) }  # => "Fi"
File.open('t.txt') {|f| f.gets(3) }  # => "Fir"
File.open('t.txt') {|f| f.gets(4) }  # => "Firs"
# No more than one line.
File.open('t.txt') {|f| f.gets(10) } # => "First line"
File.open('t.txt') {|f| f.gets(11) } # => "First line\n"
File.open('t.txt') {|f| f.gets(12) } # => "First line\n"

# Text with 2-byte characters, which will not be split.
File.open('t.rus') {|f| f.gets(1).size } # => 1
File.open('t.rus') {|f| f.gets(2).size } # => 1
File.open('t.rus') {|f| f.gets(3).size } # => 2
File.open('t.rus') {|f| f.gets(4).size } # => 2
```

Line Separator and Line Limit

With arguments `sep` and `limit` given, combines the two behaviors:

- Returns the next line as determined by line separator `sep`.
- But returns no more bytes than are allowed by the limit.

Example:

```
File.open('t.txt') {|f| f.gets('li', 20) } # => "First li"
File.open('t.txt') {|f| f.gets('li', 2) }  # => "Fi"
```

Line Number

A readable IO stream has a non-negative integer *line number*.

The relevant methods:

- `IO#lineno` : Returns the line number.
- `IO#lineno=` : Resets and returns the line number.

Unless modified by a call to method `IO#lineno=` , the line number is the number of lines read by certain line-oriented methods, according to the given line separator `sep` :

- `IO.foreach` : Increments the line number on each call to the block.
- `IO#each_line` : Increments the line number on each call to the block.
- `IO#gets` : Increments the line number.
- `IO#readline` : Increments the line number.
- `IO#readlines` : Increments the line number for each line read.

A new stream is initially has line number zero (and position zero); method `rewind` resets the line number (and position) to zero:

```
f = File.new('t.txt')
f.lineno # => 0
f.gets   # => "First line\n"
f.lineno # => 1
f.rewind
f.lineno # => 0
f.close
```

Reading lines from a stream usually changes its line number:

```
f = File.new('t.txt', 'r')
f.lineno   # => 0
f.readline # => "This is line one.\n"
f.lineno   # => 1
f.readline # => "This is the second line.\n"
f.lineno   # => 2
f.readline # => "Here's the third line.\n"
f.lineno   # => 3
f.eof?     # => true
f.close
```

Iterating over lines in a stream usually changes its line number:

```
File.open('t.txt') do |f|
  f.each_line do |line|
    p "position=#{f.pos} eof?=#{f.eof?} lineno=#{f.lineno}"
  end
end
```

Output:

```
"position=11 eof?=false lineno=1"
"position=23 eof?=false lineno=2"
"position=24 eof?=false lineno=3"
"position=36 eof?=false lineno=4"
"position=47 eof?=true lineno=5"
```

Unlike the stream's position, the line number does not affect where the next read or write will occur:

```
f = File.new('t.txt')
f.lineno = 1000
f.lineno # => 1000
f.gets   # => "First line\n"
f.lineno # => 1001
f.close
```

Associated with the line number is the global variable `$.` :

- When a stream is opened, `$.` is not set; its value is left over from previous activity in the process:

```
$. = 41
f = File.new('t.txt')
$. = 41
# => 41
f.close
```

- When a stream is read, `#.` is set to the line number for that stream:

```
f0 = File.new('t.txt')
f1 = File.new('t.dat')
f0.readlines # => ["First line\n", "Second line\n", "\n", "Fourth lin
$.           # => 5
f1.readlines # => ["\xFE\xFF\x99\x90\x99\x91\x99\x92\x99\x93\x99\x94"
$.           # => 1
f0.close
f1.close
```

- Methods `IO#rewind` and `IO#seek` do not affect `$.` :

```
f = File.new('t.txt')
f.readlines # => ["First line\n", "Second line\n", "\n", "Fourth line
$.          # => 5
f.rewind
f.seek(0, :SET)
$.          # => 5
f.close
```

Character IO

You can process an IO stream character-by-character using these methods:

- `IO#getc` : Reads and returns the next character from the stream.
- `IO#readchar` : Like `getc` , but raises an exception at end-of-stream.
- `IO#ungetc` : Pushes back (“unshifts”) a character or integer onto the stream.
- `IO#putc` : Writes a character to the stream.
- `IO#each_char` : Reads each remaining character in the stream, passing the character to the given block.

Byte IO

You can process an IO stream byte-by-byte using these methods:

- `IO#getbyte` : Returns the next 8-bit byte as an integer in range 0..255.
- `IO#readbyte` : Like `getbyte` , but raises an exception if at end-of-stream.
- `IO#ungetbyte` : Pushes back (“unshifts”) a byte back onto the stream.
- `IO#each_byte` : Reads each remaining byte in the stream, passing the byte to the given block.

Codepoint IO

You can process an IO stream codepoint-by-codepoint:

- `IO#each_codepoint` : Reads each remaining codepoint, passing it to the given block.

What’s Here

First, what’s elsewhere. Class IO:

- Inherits from class Object.
- Includes module Enumerable, which provides dozens of additional methods.

Here, class IO provides methods that are useful for:

- Creating
- Reading
- Writing
- Positioning
- Iterating

- Settings
- Querying
- Buffering
- Low-Level Access
- Other

Creating

- `::new` (aliased as `::for_fd`): Creates and returns a new IO object for the given integer file descriptor.
- `::open`: Creates a new IO object.
- `::pipe`: Creates a connected pair of reader and writer IO objects.
- `::popen`: Creates an IO object to interact with a subprocess.
- `::select`: Selects which given IO instances are ready for reading, writing, or have pending exceptions.

Reading

- `::binread`: Returns a binary string with all or a subset of bytes from the given file.
- `::read`: Returns a string with all or a subset of bytes from the given file.
- `::readlines`: Returns an array of strings, which are the lines from the given file.
- `getbyte`: Returns the next 8-bit byte read from `self` as an integer.
- `getc`: Returns the next character read from `self` as a string.
- `gets`: Returns the line read from `self`.
- `pread`: Returns all or the next *n* bytes read from `self`, not updating the receiver's offset.
- `read`: Returns all remaining or the next *n* bytes read from `self` for a given *n*.
- `read_nonblock`: the next *n* bytes read from `self` for a given *n*, in non-block mode.
- `readbyte`: Returns the next byte read from `self`; same as `getbyte`, but raises an exception on end-of-stream.
- `readchar`: Returns the next character read from `self`; same as `getc`, but raises an exception on end-of-stream.
- `readline`: Returns the next line read from `self`; same as getline, but raises an exception of end-of-stream.

- readlines : Returns an array of all lines read read from self .
- readpartial : Returns up to the given number of bytes from self .

Writing

- ::binwrite : Writes the given string to the file at the given filepath, in binary mode.
- ::write : Writes the given string to self .
- << : Appends the given string to self .
- print : Prints last read line or given objects to self .
- printf : Writes to self based on the given format string and objects.
- putc : Writes a character to self .
- puts : Writes lines to self , making sure line ends with a newline.
- pwrite : Writes the given string at the given offset, not updating the receiver's offset.
- write : Writes one or more given strings to self .
- write_nonblock : Writes one or more given strings to self in non-blocking mode.

Positioning

- lineno : Returns the current line number in self .
- lineno= : Sets the line number is self .
- pos (aliased as tell): Returns the current byte offset in self .
- pos= : Sets the byte offset in self .
- reopen : Reassociates self with a new or existing IO stream.
- rewind : Positions self to the beginning of input.
- seek : Sets the offset for self relative to given position.

Iterating

- ::foreach : Yields each line of given file to the block.
- each (aliased as each_line): Calls the given block with each successive line in self .
- each_byte : Calls the given block with each successive byte in self as an integer.
- each_char : Calls the given block with each successive character in self as a string.

- each_codepoint : Calls the given block with each successive codepoint in `self` as an integer.

Settings

- autoclose= : Sets whether `self` auto-closes.
- binmode : Sets `self` to binary mode.
- close : Closes `self`.
- close_on_exec= : Sets the close-on-exec flag.
- close_read : Closes `self` for reading.
- close_write : Closes `self` for writing.
- set_encoding : Sets the encoding for `self`.
- set_encoding_by_bom : Sets the encoding for `self`, based on its Unicode byte-order-mark.
- sync= : Sets the sync-mode to the given value.

Querying

- autoclose? : Returns whether `self` auto-closes.
- binmode? : Returns whether `self` is in binary mode.
- close_on_exec? : Returns the close-on-exec flag for `self`.
- closed? : Returns whether `self` is closed.
- eof? (aliased as eof): Returns whether `self` is at end-of-stream.
- external_encoding : Returns the external encoding object for `self`.
- fileno (aliased as to_i): Returns the integer file descriptor for `self`
- internal_encoding : Returns the internal encoding object for `self`.
- pid : Returns the process ID of a child process associated with `self`, if `self` was created by ::popen.
- stat : Returns the File::Stat object containing status information for `self`.
- sync : Returns whether `self` is in sync-mode.
- tty? (aliased as isatty): Returns whether `self` is a terminal.

Buffering

- fdatasync : Immediately writes all buffered data in `self` to disk.

- `flush` : Flushes any buffered data within `self` to the underlying operating system.
- `fsync` : Immediately writes all buffered data and attributes in `self` to disk.
- `ungetbyte` : Prepends buffer for `self` with given integer byte or string.
- `ungetc` : Prepends buffer for `self` with given string.

Low-Level Access

- `::sysopen` : Opens the file given by its path, returning the integer file descriptor.
- `advise` : Announces the intention to access data from `self` in a specific way.
- `fcntl` : Passes a low-level command to the file specified by the given file descriptor.
- `ioctl` : Passes a low-level command to the device specified by the given file descriptor.
- `sysread` : Returns up to the next *n* bytes read from self using a low-level read.
- `sysseek` : Sets the offset for `self` .
- `syswrite` : Writes the given string to `self` using a low-level write.

Other

- `::copy_stream` : Copies data from a source to a destination, each of which is a filepath or an IO-like object.
- `::try_convert` : Returns a new IO object resulting from converting the given object.
- `inspect` : Returns the string representation of `self` .

Constants

EWOULDBLOCKWaitReadable

EAGAINWaitReadable

EWOULDBLOCKWaitWritable

EAGAINWaitWritable

PRIORITY

READABLE

SEEK_CUR

Set I/O position from the current position

SEEK_DATA

Set I/O position to the next location containing data

SEEK_END

Set I/O position from the end

SEEK_HOLE

Set I/O position to the next hole

SEEK_SET

Set I/O position from the beginning

WRITABLE

Public Class Methods

binread(path, length = nil, offset = 0) → string or nil

Behaves like `IO.read`, except that the stream is opened in binary mode with ASCII-8BIT encoding.

When called from class IO (but not subclasses of IO), this method has potential security vulnerabilities if called with untrusted input; see Command Injection.

binwrite(path, string, offset = 0) → integer

Behaves like `IO.write`, except that the stream is opened in binary mode with ASCII-8BIT encoding.

When called from class IO (but not subclasses of IO), this method has potential security vulnerabilities if called with untrusted input; see Command Injection.

copy_stream(src, dst, src_length = nil, src_offset = 0) → integer

Copies from the given `src` to the given `dst`, returning the number of bytes copied.

- The given `src` must be one of the following:
  - The path to a readable file, from which source data is to be read.
  - An IO-like object, opened for reading and capable of responding to method `:readpartial` or method `:read`.
- The given `dst` must be one of the following:
  - The path to a writable file, to which data is to be written.
  - An IO-like object, opened for writing and capable of responding to method `:write`.

The examples here use file `t.txt` as source:

```
File.read('t.txt')
# => "First line\nSecond line\n\nThird line\nFourth line\n"
File.read('t.txt').size # => 47
```

If only arguments `src` and `dst` are given, the entire source stream is copied:

```
# Paths.
IO.copy_stream('t.txt', 't.tmp')  # => 47

# IOs (recall that a File is also an IO).
src_io = File.open('t.txt', 'r') # => #<File:t.txt>
dst_io = File.open('t.tmp', 'w') # => #<File:t.tmp>
IO.copy_stream(src_io, dst_io)   # => 47
src_io.close
dst_io.close
```

With argument `src_length` a non-negative integer, no more than that many bytes are copied:

```
IO.copy_stream('t.txt', 't.tmp', 10) # => 10
File.read('t.tmp')                   # => "First line"
```

With argument `src_offset` also given, the source stream is read beginning at that offset:

```
IO.copy_stream('t.txt', 't.tmp', 11, 11) # => 11
IO.read('t.tmp')                         # => "Second line"
```

`for_fd(fd, mode = 'r', **opts) → io`

Synonym for `IO.new`.

foreach(path, sep = $/, **opts) {|line| block } → nil
foreach(path, limit, **opts) {|line| block } → nil
foreach(path, sep, limit, **opts) {|line| block } → nil
foreach(...) → an_enumerator

Calls the block with each successive line read from the stream.

When called from class IO (but not subclasses of IO), this method has potential security vulnerabilities if called with untrusted input; see Command Injection.

The first argument must be a string that is the path to a file.

With only argument `path` given, parses lines from the file at the given `path`, as determined by the default line separator, and calls the block with each successive line:

```
File.foreach('t.txt') {|line| p line }
```

Output: the same as above.

For both forms, command and path, the remaining arguments are the same.

With argument `sep` given, parses lines as determined by that line separator (see Line Separator):

```
File.foreach('t.txt', 'li') {|line| p line }
```

Output:

```
"First li"
"ne\nSecond li"
"ne\n\nThird li"
"ne\nFourth li"
"ne\n"
```

Each paragraph:

```
File.foreach('t.txt', '') {|paragraph| p paragraph }
```

Output:

```
"First line\nSecond line\n\n"
"Third line\nFourth line\n"
```

With argument `limit` given, parses lines as determined by the default line separator and the given line-length limit (see Line Limit):

```
File.foreach('t.txt', 7) {|line| p line }
```

Output:

```
"First l"
"ine\n"
"Second "
"line\n"
"\n"
"Third l"
"ine\n"
"Fourth l"
"line\n"
```

With arguments `sep` and `limit` given, parses lines as determined by the given line separator and the given line-length limit (see Line Separator and Line Limit):

Optional keyword arguments `opts` specify:

- Open Options.
- Encoding options.
- Line Options.

Returns an `Enumerator` if no block is given.

new(fd, mode = 'r', **opts) → io

Creates and returns a new IO object (file stream) from a file descriptor.

IO.new may be useful for interaction with low-level libraries. For higher-level interactions, it may be simpler to create the file stream using `File.open`.

Argument `fd` must be a valid file descriptor (integer):

```
path = 't.tmp'
fd = IO.sysopen(path) # => 3
IO.new(fd)            # => #<IO:fd 3>
```

The new IO object does not inherit encoding (because the integer file descriptor does not have an encoding):

```
fd = IO.sysopen('t.rus', 'rb')
io = IO.new(fd)
io.external_encoding # => #<Encoding:UTF-8> # Not ASCII-8BIT.
```

Optional argument `mode` (defaults to 'r') must specify a valid mode; see Access Modes:

```
IO.new(fd, 'w')          # => #<IO:fd 3>
IO.new(fd, File::WRONLY) # => #<IO:fd 3>
```

Optional keyword arguments `opts` specify:

- Open Options.

- Encoding options.

Examples:

```
IO.new(fd, internal_encoding: nil) # => #<IO:fd 3>
IO.new(fd, autoclose: true)        # => #<IO:fd 3>
```

open(fd, mode = 'r', **opts) → io
open(fd, mode = 'r', **opts) {|io| ... } → object

Creates a new IO object, via `IO.new` with the given arguments.

With no block given, returns the IO object.

With a block given, calls the block with the IO object and returns the block's value.

pipe(opts) → [read_io, write_io]**
pipe(enc, **opts) → [read_io, write_io]
pipe(ext_enc, int_enc, **opts) → [read_io, write_io]
pipe(opts) {|read_io, write_io| ...} → object**
pipe(enc, **opts) {|read_io, write_io| ...} → object
pipe(ext_enc, int_enc, **opts) {|read_io, write_io| ...} → object

Creates a pair of pipe endpoints, `read_io` and `write_io`, connected to each other.

If argument `enc_string` is given, it must be a string containing one of:

- The name of the encoding to be used as the external encoding.
- The colon-separated names of two encodings to be used as the external and internal encodings.

If argument `int_enc` is given, it must be an `Encoding` object or encoding name string that specifies the internal encoding to be used; if argument `ext_enc` is also given, it must be an `Encoding` object or encoding name string that specifies the external encoding to be used.

The string read from `read_io` is tagged with the external encoding; if an internal encoding is also specified, the string is converted to, and tagged with, that encoding.

If any encoding is specified, optional hash arguments specify the conversion option.

Optional keyword arguments `opts` specify:

- Open Options.
- Encoding Options.

With no block given, returns the two endpoints in an array:

```
IO.pipe # => [#<IO:fd 4>, #<IO:fd 5>]
```

With a block given, calls the block with the two endpoints; closes both endpoints and returns the value of the block:

```
IO.pipe {|read_io, write_io| p read_io; p write_io }
```

Output:

```
#<IO:fd 6>
#<IO:fd 7>
```

Not available on all platforms.

In the example below, the two processes close the ends of the pipe that they are not using. This is not just a cosmetic nicety. The read end of a pipe will not generate an end of file condition if there are any writers with the pipe still open. In the case of the parent process, the `rd.read` will never return if it does not first issue a `wr.close`:

```
rd, wr = IO.pipe

if fork
  wr.close
  puts "Parent got: <#{rd.read}>"
  rd.close
  Process.wait
else
  rd.close
  puts 'Sending message to parent'
  wr.write "Hi Dad"
  wr.close
end
```

produces:

```
Sending message to parent
Parent got: <Hi Dad>
```

`popen(env = {}, cmd, mode = 'r', **opts) → io`
`popen(env = {}, cmd, mode = 'r', **opts) {|io| ... } → object`

Executes the given command `cmd` as a subprocess whose $stdin and $stdout are connected to a new stream `io`.

This method has potential security vulnerabilities if called with untrusted input; see Command Injection.

If no block is given, returns the new stream, which depending on given `mode` may be open for reading, writing, or both. The stream should be explicitly closed (eventually) to avoid resource leaks.

If a block is given, the stream is passed to the block (again, open for reading, writing, or both); when the block exits, the stream is closed, and the block's value is assigned to global variable `$?` and returned.

Optional argument `mode` may be any valid IO mode. See Access Modes.

Required argument `cmd` determines which of the following occurs:

- The process forks.
- A specified program runs in a shell.
- A specified program runs with specified arguments.
- A specified program runs with specified arguments and a specified `argv0`.

Each of these is detailed below.

The optional hash argument `env` specifies name/value pairs that are to be added to the environment variables for the subprocess:

```
IO.popen({'FOO' => 'bar'}, 'ruby', 'r+') do |pipe|
  pipe.puts 'puts ENV["FOO"]'
  pipe.close_write
  pipe.gets
end => "bar\n"
```

Optional keyword arguments `opts` specify:

- Open options.
- Encoding options.
- Options for `Kernel#spawn`.

Forked Process

When argument `cmd` is the 1-character string `'-'`, causes the process to fork:

```
IO.popen('-') do |pipe|
  if pipe
    $stderr.puts "In parent, child pid is #{pipe.pid}\n"
  else
    $stderr.puts "In child, pid is #{$$}\n"
  end
end
```

Output:

```
In parent, child pid is 26253
In child, pid is 26253
```

Note that this is not supported on all platforms.

Shell Subprocess

When argument `cmd` is a single string (but not `'-'`), the program named `cmd` is run as a shell command:

```
IO.popen('uname') do |pipe|
  pipe.readlines
end
```

Output:

```
["Linux\n"]
```

Another example:

```
IO.popen('/bin/sh', 'r+') do |pipe|
  pipe.puts('ls')
  pipe.close_write
  $stderr.puts pipe.readlines.size
end
```

Output:

```
213
```

Program Subprocess

When argument `cmd` is an array of strings, the program named `cmd[0]` is run with all elements of `cmd` as its arguments:

```
IO.popen(['du', '..', '.']) do |pipe|
  $stderr.puts pipe.readlines.size
end
```

Output:

```
1111
```

Program Subprocess with `argv0`

When argument `cmd` is an array whose first element is a 2-element string array and whose remaining elements (if any) are strings:

- `cmd[0][0]` (the first string in the nested array) is the name of a program that is run.
- `cmd[0][1]` (the second string in the nested array) is set as the program's `argv[0]`.

- `cmd[1..-1]` (the strings in the outer array) are the program's arguments.

Example (sets `$0` to 'foo'):

```
IO.popen([['/bin/sh', 'foo'], '-c', 'echo $0']).read # => "foo\n"
```

Some Special Examples

```
# Set IO encoding.
IO.popen("nkf -e filename", :external_encoding=>"EUC-JP") {|nkf_io|
  euc_jp_string = nkf_io.read
}

# Merge standard output and standard error using Kernel#spawn option. See Ker
IO.popen(["ls", "/", :err=>[:child, :out]]) do |io|
  ls_result_with_error = io.read
end

# Use mixture of spawn options and IO options.
IO.popen(["ls", "/"], :err=>[:child, :out]) do |io|
  ls_result_with_error = io.read
end

 f = IO.popen("uname")
 p f.readlines
 f.close
 puts "Parent is #{Process.pid}"
 IO.popen("date") {|f| puts f.gets }
 IO.popen("-") {|f| $stderr.puts "#{Process.pid} is here, f is #{f.inspect}"}
 p $?
 IO.popen(%w"sed -e s|^|<foo>| -e s&$&;zot;&", "r+") {|f|
   f.puts "bar"; f.close_write; puts f.gets
 }
```

Output (from last section):

```
["Linux\n"]
Parent is 21346
Thu Jan 15 22:41:19 JST 2009
21346 is here, f is #<IO:fd 3>
21352 is here, f is nil
#<Process::Status: pid 21352 exit 0>
<foo>bar;zot;
```

Raises exceptions that `IO.pipe` and `Kernel.spawn` raise.

`read(path, length = nil, offset = 0, **opts) → string or nil`

Opens the stream, reads and returns some or all of its content, and closes the stream; returns `nil` if no bytes were read.

When called from class IO (but not subclasses of IO), this method has potential security vulnerabilities if called with untrusted input; see Command Injection.

The first argument must be a string that is the path to a file.

With only argument `path` given, reads in text mode and returns the entire content of the file at the given path:

```
IO.read('t.txt')
# => "First line\nSecond line\n\nThird line\nFourth line\n"
```

On Windows, text mode can terminate reading and leave bytes in the file unread when encountering certain special bytes. Consider using `IO.binread` if all bytes in the file should be read.

With argument `length`, returns `length` bytes if available:

```
IO.read('t.txt', 7) # => "First l"
IO.read('t.txt', 700)
# => "First line\r\nSecond line\r\n\r\nFourth line\r\nFifth line\r\n"
```

With arguments `length` and `offset`, returns `length` bytes if available, beginning at the given `offset`:

```
IO.read('t.txt', 10, 2)   # => "rst line\nS"
IO.read('t.txt', 10, 200) # => nil
```

Optional keyword arguments `opts` specify:

- Open Options.
- Encoding options.

`readlines(path, sep = $/, **opts) → array`
`readlines(path, limit, **opts) → array`
`readlines(path, sep, limit, **opts) → array`

Returns an array of all lines read from the stream.

When called from class IO (but not subclasses of IO), this method has potential security vulnerabilities if called with untrusted input; see Command Injection.

The first argument must be a string that is the path to a file.

With only argument `path` given, parses lines from the file at the given `path`, as determined by the default line separator, and returns those lines in an array:

```
IO.readlines('t.txt')
# => ["First line\n", "Second line\n", "\n", "Third line\n", "Fourth line\n"]
```

With argument `sep` given, parses lines as determined by that line separator (see Line Separator):

```
# Ordinary separator.
IO.readlines('t.txt', 'li')
# =>["First li", "ne\nSecond li", "ne\n\nThird li", "ne\nFourth li", "ne\n"]
# Get-paragraphs separator.
IO.readlines('t.txt', '')
# => ["First line\nSecond line\n\n", "Third line\nFourth line\n"]
# Get-all separator.
IO.readlines('t.txt', nil)
# => ["First line\nSecond line\n\nThird line\nFourth line\n"]
```

With argument `limit` given, parses lines as determined by the default line separator and the given line-length limit (see Line Limit):

```
IO.readlines('t.txt', 7)
# => ["First l", "ine\n", "Second ", "line\n", "\n", "Third l", "ine\n", "Fou
```

With arguments `sep` and `limit` given, parses lines as determined by the given line separator and the given line-length limit (see Line Separator and Line Limit):

Optional keyword arguments `opts` specify:

- Open Options.
- Encoding options.
- Line Options.

`select(read_ios, write_ios = [], error_ios = [], timeout = nil) → array or nil`

Invokes system call select(2), which monitors multiple file descriptors, waiting until one or more of the file descriptors becomes ready for some class of I/O operation.

Not implemented on all platforms.

Each of the arguments `read_ios`, `write_ios`, and `error_ios` is an array of `IO` objects.

Argument `timeout` is an integer timeout interval in seconds.

The method monitors the IO objects given in all three arrays, waiting for some to be ready; returns a 3-element array whose elements are:

- An array of the objects in `read_ios` that are ready for reading.
- An array of the objects in `write_ios` that are ready for writing.
- An array of the objects in `error_ios` have pending exceptions.

If no object becomes ready within the given `timeout`, `nil` is returned.

IO.select peeks the buffer of IO objects for testing readability. If the IO buffer is not empty, IO.select immediately notifies readability. This “peek” only happens for IO objects. It does not happen for IO-like objects such as OpenSSL::SSL::SSLSocket.

The best way to use IO.select is invoking it after non-blocking methods such as `read_nonblock`, `write_nonblock`, etc. The methods raise an exception which is extended by `IO::WaitReadable` or `IO::WaitWritable`. The modules notify how the caller should wait with IO.select. If `IO::WaitReadable` is raised, the caller should wait for reading. If `IO::WaitWritable` is raised, the caller should wait for writing.

So, blocking read (`readpartial`) can be emulated using `read_nonblock` and IO.select as follows:

```
begin
  result = io_like.read_nonblock(maxlen)
rescue IO::WaitReadable
  IO.select([io_like])
  retry
rescue IO::WaitWritable
  IO.select(nil, [io_like])
  retry
end
```

Especially, the combination of non-blocking methods and IO.select is preferred for `IO` like objects such as OpenSSL::SSL::SSLSocket. It has `to_io` method to return underlying `IO` object. `IO.select` calls `to_io` to obtain the file descriptor to wait.

This means that readability notified by IO.select doesn’t mean readability from OpenSSL::SSL::SSLSocket object.

The most likely situation is that OpenSSL::SSL::SSLSocket buffers some data. IO.select doesn’t see the buffer. So IO.select can block when OpenSSL::SSL::SSLSocket#readpartial doesn’t block.

However, several more complicated situations exist.

SSL is a protocol which is sequence of records. The record consists of multiple bytes. So, the remote side of SSL sends a partial record, `IO.select` notifies readability but OpenSSL::SSL::SSLSocket cannot decrypt a byte and OpenSSL::SSL::SSLSocket#readpartial will block.

Also, the remote side can request SSL renegotiation which forces the local SSL engine to write some data. This means OpenSSL::SSL::SSLSocket#readpartial may invoke `write` system call and it can block. In such a situation, OpenSSL::SSL::SSLSocket#read_nonblock raises `IO::WaitWritable` instead of blocking. So, the caller should wait for ready for writability as above example.

The combination of non-blocking methods and IO.select is also useful for streams such as tty, pipe socket socket when multiple processes read from a stream.

Finally, Linux kernel developers don’t guarantee that readability of select(2) means readability of following read(2) even for a single process; see select(2)

Invoking IO.select before `IO#readpartial` works well as usual. However it is not the best way to use IO.select.

The writability notified by select(2) doesn't show how many bytes are writable. `IO#write` method blocks until given whole string is written. So, `IO#write(two or more bytes)` can block after writability is notified by IO.select. `IO#write_nonblock` is required to avoid the blocking.

Blocking write (`write`) can be emulated using `write_nonblock` and `IO.select` as follows: `IO::WaitReadable` should also be rescued for SSL renegotiation in OpenSSL::SSL::SSLSocket.

```
while 0 < string.bytesize
  begin
    written = io_like.write_nonblock(string)
  rescue IO::WaitReadable
    IO.select([io_like])
    retry
  rescue IO::WaitWritable
    IO.select(nil, [io_like])
    retry
  end
  string = string.byteslice(written..-1)
end
```

Example:

```
rp, wp = IO.pipe
mesg = "ping "
100.times {
  # IO.select follows IO#read.  Not the best way to use IO.select.
  rs, ws, = IO.select([rp], [wp])
  if r = rs[0]
    ret = r.read(5)
    print ret
    case ret
    when /ping/
      mesg = "pong\n"
    when /pong/
      mesg = "ping "
    end
  end
  if w = ws[0]
    w.write(mesg)
  end
}
```

Output:

```
ping pong
ping pong
ping pong
(snipped)
ping
```

sysopen(path, mode = 'r', perm = 0666) → integer

Opens the file at the given path with the given mode and permissions; returns the integer file descriptor.

If the file is to be readable, it must exist; if the file is to be writable and does not exist, it is created with the given permissions:

```
File.write('t.tmp', '')  # => 0
IO.sysopen('t.tmp')      # => 8
IO.sysopen('t.tmp', 'w') # => 9
```

try_convert(object) → new_io or nil

Attempts to convert `object` into an IO object via method `to_io`; returns the new IO object if successful, or `nil` otherwise:

```
IO.try_convert(STDOUT)   # => #<IO:<STDOUT>>
IO.try_convert(ARGF)     # => #<IO:<STDIN>>
IO.try_convert('STDOUT') # => nil
```

write(path, data, offset = 0, **opts) → integer

Opens the stream, writes the given `data` to it, and closes the stream; returns the number of bytes written.

When called from class IO (but not subclasses of IO), this method has potential security vulnerabilities if called with untrusted input; see Command Injection.

The first argument must be a string that is the path to a file.

With only argument `path` given, writes the given `data` to the file at that path:

```
IO.write('t.tmp', 'abc')    # => 3
File.read('t.tmp')          # => "abc"
```

If `offset` is zero (the default), the file is overwritten:

```
IO.write('t.tmp', 'A')      # => 1
File.read('t.tmp')          # => "A"
```

If `offset` in within the file content, the file is partly overwritten:

```
IO.write('t.tmp', 'abcdef') # => 3
File.read('t.tmp')          # => "abcdef"
# Offset within content.
IO.write('t.tmp', '012', 2) # => 3
File.read('t.tmp')          # => "ab012f"
```

If `offset` is outside the file content, the file is padded with null characters `"\u0000"`:

```
IO.write('t.tmp', 'xyz', 10) # => 3
File.read('t.tmp')           # => "ab012f\u0000\u0000\u0000\u0000xyz"
```

Optional keyword arguments `opts` specify:

- Open Options.
- Encoding options.

Public Instance Methods

`self << object → self`

Writes the given `object` to `self`, which must be opened for writing (see Access Modes); returns `self`; if `object` is not a string, it is converted via method `to_s`:

```
$stdout << 'Hello' << ', ' << 'World!' << "\n"
$stdout << 'foo' << :bar << 2 << "\n"
```

Output:

```
Hello, World!
foobar2
```

`advise(advice, offset = 0, len = 0) → nil`

Invokes Posix system call posix_fadvise(2), which announces an intention to access data from the current file in a particular manner.

The arguments and results are platform-dependent.

The relevant data is specified by:

- `offset`: The offset of the first byte of data.
- `len`: The number of bytes to be accessed; if `len` is zero, or is larger than the number of bytes remaining, all remaining bytes will be accessed.

Argument `advice` is one of the following symbols:

- `:normal`: The application has no advice to give about its access pattern for the specified data. If no advice is given for an open file, this is the default assumption.

- `:sequential` : The application expects to access the specified data sequentially (with lower offsets read before higher ones).
- `:random` : The specified data will be accessed in random order.
- `:noreuse` : The specified data will be accessed only once.
- `:willneed` : The specified data will be accessed in the near future.
- `:dontneed` : The specified data will not be accessed in the near future.

Not implemented on all platforms.

autoclose = bool → true or false

Sets auto-close flag.

```
f = File.open(File::NULL)
IO.for_fd(f.fileno).close
f.gets # raises Errno::EBADF

f = File.open(File::NULL)
g = IO.for_fd(f.fileno)
g.autoclose = false
g.close
f.gets # won't cause Errno::EBADF
```

autoclose? → true or false

Returns `true` if the underlying file descriptor of *ios* will be closed at its finalization or at calling `close`, otherwise `false`.

binmode → self

Sets the stream's data mode as binary (see Data Mode).

A stream's data mode may not be changed from binary to text.

binmode? → true or false

Returns `true` if the stream is on binary mode, `false` otherwise. See Data Mode.

close → nil

Closes the stream for both reading and writing if open for either or both; returns `nil`. See Open and Closed Streams.

If the stream is open for writing, flushes any buffered writes to the operating system before closing.

If the stream was opened by `IO.popen`, sets global variable `$?` (child exit status).

Example:

```
IO.popen('ruby', 'r+') do |pipe|
  puts pipe.closed?
  pipe.close
  puts $?
  puts pipe.closed?
end
```

Output:

```
false
pid 13760 exit 0
true
```

Related: `IO#close_read`, `IO#close_write`, `IO#closed?`.

close_on_exec = bool → true or false

Sets a close-on-exec flag.

```
f = File.open(File::NULL)
f.close_on_exec = true
system("cat", "/proc/self/fd/#{f.fileno}") # cat: /proc/self/fd/3: No such fi
f.closed?                 #=> false
```

Ruby sets close-on-exec flags of all file descriptors by default since Ruby 2.0.0. So you don't need to set by yourself. Also, unsetting a close-on-exec flag can cause file descriptor leak if another thread use fork() and exec() (via system() method for example). If you really needs file descriptor inheritance to child process, use spawn()'s argument such as fd=>fd.

close_on_exec? → true or false

Returns `true` if the stream will be closed on exec, `false` otherwise:

```
f = File.open('t.txt')
f.close_on_exec? # => true
f.close_on_exec = false
f.close_on_exec? # => false
f.close
```

close_read → nil

Closes the stream for reading if open for reading; returns `nil`. See Open and Closed Streams.

If the stream was opened by `IO.popen` and is also closed for writing, sets global variable `$?` (child exit status).

Example:

```
IO.popen('ruby', 'r+') do |pipe|
  puts pipe.closed?
  pipe.close_write
  puts pipe.closed?
  pipe.close_read
  puts $?
  puts pipe.closed?
end
```

Output:

```
false
false
pid 14748 exit 0
true
```

Related: `IO#close`, `IO#close_write`, `IO#closed?`.

close_write → nil

Closes the stream for writing if open for writing; returns `nil`. See Open and Closed Streams.

Flushes any buffered writes to the operating system before closing.

If the stream was opened by `IO.popen` and is also closed for reading, sets global variable `$?` (child exit status).

```
IO.popen('ruby', 'r+') do |pipe|
  puts pipe.closed?
  pipe.close_read
  puts pipe.closed?
  pipe.close_write
  puts $?
  puts pipe.closed?
end
```

Output:

```
false
false
pid 15044 exit 0
true
```

Related: IO#close, IO#close_read, IO#closed?.

closed? → true or false

Returns `true` if the stream is closed for both reading and writing, `false` otherwise. See Open and Closed Streams.

```
IO.popen('ruby', 'r+') do |pipe|
  puts pipe.closed?
  pipe.close_read
  puts pipe.closed?
  pipe.close_write
  puts pipe.closed?
end
```

Output:

```
false
false
true
```

Related: IO#close_read, IO#close_write, IO#close.

each -> enumerator

Calls the block with each remaining line read from the stream; returns `self`. Does nothing if already at end-of-stream; See Line IO.

With no arguments given, reads lines as determined by line separator `$/`:

```
f = File.new('t.txt')
f.each_line {|line| p line }
f.each_line {|line| fail 'Cannot happen' }
f.close
```

Output:

```
"First line\n"
"Second line\n"
"\n"
"Fourth line\n"
"Fifth line\n"
```

With only string argument `sep` given, reads lines as determined by line separator `sep`; see Line Separator:

```
f = File.new('t.txt')
f.each_line('li') {|line| p line }
```

```
f.close
```

Output:

```
"First li"
"ne\nSecond li"
"ne\n\nFourth li"
"ne\nFifth li"
"ne\n"
```

The two special values for `sep` are honored:

```
f = File.new('t.txt')
# Get all into one string.
f.each_line(nil) {|line| p line }
f.close
```

Output:

```
"First line\nSecond line\n\nFourth line\nFifth line\n"

f.rewind
# Get paragraphs (up to two line separators).
f.each_line('') {|line| p line }
```

Output:

```
"First line\nSecond line\n\n"
"Fourth line\nFifth line\n"
```

With only integer argument `limit` given, limits the number of bytes in each line; see Line Limit:

```
f = File.new('t.txt')
f.each_line(8) {|line| p line }
f.close
```

Output:

```
"First li"
"ne\n"
"Second l"
"ine\n"
"\n"
"Fourth l"
"ine\n"
"Fifth li"
"ne\n"
```

With arguments `sep` and `limit` given, combines the two behaviors:

- Calls with the next line as determined by line separator `sep`.
- But returns no more bytes than are allowed by the limit.

Optional keyword argument `chomp` specifies whether line separators are to be omitted:

```
f = File.new('t.txt')
f.each_line(chomp: true) {|line| p line }
f.close
```

Output:

```
"First line"
"Second line"
""
"Fourth line"
"Fifth line"
```

Returns an `Enumerator` if no block is given.

Also aliased as: each_line

each_byte {|byte| ... } → self
each_byte → enumerator

Calls the given block with each byte (0..255) in the stream; returns `self`. See Byte IO.

```
f = File.new('t.rus')
a = []
f.each_byte {|b| a << b }
a # => [209, 130, 208, 181, 209, 129, 209, 130]
f.close
```

Returns an `Enumerator` if no block is given.

Related: `IO#each_char`, `IO#each_codepoint`.

each_char {|c| ... } → self
each_char → enumerator

Calls the given block with each character in the stream; returns `self`. See Character IO.

```
f = File.new('t.rus')
a = []
f.each_char {|c| a << c.ord }
a # => [1090, 1077, 1089, 1090]
f.close
```

Returns an `Enumerator` if no block is given.

Related: `IO#each_byte`, `IO#each_codepoint`.

each_codepoint {|c| ... } → self
each_codepoint → enumerator

Calls the given block with each codepoint in the stream; returns `self`:

```
f = File.new('t.rus')
a = []
f.each_codepoint {|c| a << c }
a # => [1090, 1077, 1089, 1090]
f.close
```

Returns an `Enumerator` if no block is given.

Related: `IO#each_byte`, `IO#each_char`.

each_line(*args)

Calls the block with each remaining line read from the stream; returns `self`. Does nothing if already at end-of-stream; See Line IO.

With no arguments given, reads lines as determined by line separator `$/`:

```
f = File.new('t.txt')
f.each_line {|line| p line }
f.each_line {|line| fail 'Cannot happen' }
f.close
```

Output:

```
"First line\n"
"Second line\n"
"\n"
"Fourth line\n"
"Fifth line\n"
```

With only string argument `sep` given, reads lines as determined by line separator `sep`; see Line Separator:

```
f = File.new('t.txt')
f.each_line('li') {|line| p line }
f.close
```

Output:

```
"First li"
"ne\nSecond li"
```

```
"ne\n\nFourth li"
"ne\nFifth li"
"ne\n"
```

The two special values for `sep` are honored:

```
f = File.new('t.txt')
# Get all into one string.
f.each_line(nil) {|line| p line }
f.close
```

Output:

```
"First line\nSecond line\n\nFourth line\nFifth line\n"

f.rewind
# Get paragraphs (up to two line separators).
f.each_line('') {|line| p line }
```

Output:

```
"First line\nSecond line\n\n"
"Fourth line\nFifth line\n"
```

With only integer argument `limit` given, limits the number of bytes in each line; see Line Limit:

```
f = File.new('t.txt')
f.each_line(8) {|line| p line }
f.close
```

Output:

```
"First li"
"ne\n"
"Second l"
"ine\n"
"\n"
"Fourth l"
"ine\n"
"Fifth li"
"ne\n"
```

With arguments `sep` and `limit` given, combines the two behaviors:

- Calls with the next line as determined by line separator `sep`.
- But returns no more bytes than are allowed by the limit.

Optional keyword argument `chomp` specifies whether line separators are to be omitted:

```
f = File.new('t.txt')
f.each_line(chomp: true) {|line| p line }
f.close
```

Output:

```
"First line"
"Second line"
""
"Fourth line"
"Fifth line"
```

Returns an `Enumerator` if no block is given.

Alias for: each

eof → true or false

Returns `true` if the stream is positioned at its end, `false` otherwise; see Position:

```
f = File.open('t.txt')
f.eof           # => false
f.seek(0, :END) # => 0
f.eof           # => true
f.close
```

Raises an exception unless the stream is opened for reading; see Mode.

If `self` is a stream such as pipe or socket, this method blocks until the other end sends some data or closes it:

```
r, w = IO.pipe
Thread.new { sleep 1; w.close }
r.eof? # => true # After 1-second wait.

r, w = IO.pipe
Thread.new { sleep 1; w.puts "a" }
r.eof?  # => false # After 1-second wait.

r, w = IO.pipe
r.eof?  # blocks forever
```

Note that this method reads data to the input byte buffer. So `IO#sysread` may not behave as you intend with `IO#eof?`, unless you call `IO#rewind` first (which is not available for some streams).

Also aliased as: eof?

eof?()

Returns `true` if the stream is positioned at its end, `false` otherwise; see Position:

```
f = File.open('t.txt')
f.eof           # => false
f.seek(0, :END) # => 0
f.eof           # => true
f.close
```

Raises an exception unless the stream is opened for reading; see Mode.

If `self` is a stream such as pipe or socket, this method blocks until the other end sends some data or closes it:

```
r, w = IO.pipe
Thread.new { sleep 1; w.close }
r.eof? # => true # After 1-second wait.

r, w = IO.pipe
Thread.new { sleep 1; w.puts "a" }
r.eof?  # => false # After 1-second wait.

r, w = IO.pipe
r.eof?  # blocks forever
```

Note that this method reads data to the input byte buffer. So `IO#sysread` may not behave as you intend with `IO#eof?`, unless you call `IO#rewind` first (which is not available for some streams).

Alias for: eof

external_encoding → encoding or nil

Returns the `Encoding` object that represents the encoding of the stream, or `nil` if the stream is in write mode and no encoding is specified.

See Encodings.

fcntl(integer_cmd, argument) → integer

Invokes Posix system call fcntl(2), which provides a mechanism for issuing low-level commands to control or query a file-oriented I/O stream. Arguments and results are platform dependent.

If `argument` is a number, its value is passed directly; if it is a string, it is interpreted as a binary sequence of bytes. (`Array#pack` might be a useful way to build this string.)

Not implemented on all platforms.

fdatasync → 0

Immediately writes to disk all data buffered in the stream, via the operating system's: `fdatasync(2)`, if supported, otherwise via `fsync(2)`, if supported; otherwise raises an exception.

fileno → integer

Returns the integer file descriptor for the stream:

```
$stdin.fileno             # => 0
$stdout.fileno            # => 1
$stderr.fileno            # => 2
File.open('t.txt').fileno # => 10
f.close
```

Also aliased as: to_i

flush → self

Flushes data buffered in `self` to the operating system (but does not necessarily flush data buffered in the operating system):

```
$stdout.print 'no newline' # Not necessarily flushed.
$stdout.flush             # Flushed.
```

fsync → 0

Immediately writes to disk all data buffered in the stream, via the operating system's `fsync(2)`.

Note this difference:

- `IO#sync=`: Ensures that data is flushed from the stream's internal buffers, but does not guarantee that the operating system actually writes the data to disk.
- `IO#fsync`: Ensures both that data is flushed from internal buffers, and that data is written to disk.

Raises an exception if the operating system does not support `fsync(2)`.

getbyte → integer or nil

Reads and returns the next byte (in range 0..255) from the stream; returns `nil` if already at end-of-stream. See Byte IO.

```
f = File.open('t.txt')
f.getbyte # => 70
f.close
f = File.open('t.rus')
f.getbyte # => 209
f.close
```

Related: IO#readbyte (may raise EOFError).

getc → character or nil

Reads and returns the next 1-character string from the stream; returns `nil` if already at end-of-stream. See Character IO.

```
f = File.open('t.txt')
f.getc     # => "F"
f.close
f = File.open('t.rus')
f.getc.ord # => 1090
f.close
```

Related: IO#readchar (may raise EOFError).

gets(sep = $/, chomp: false) → string or nil
gets(limit, chomp: false) → string or nil
gets(sep, limit, chomp: false) → string or nil

Reads and returns a line from the stream; assigns the return value to `$_`. See Line IO.

With no arguments given, returns the next line as determined by line separator `$/`, or `nil` if none:

```
f = File.open('t.txt')
f.gets # => "First line\n"
$_     # => "First line\n"
f.gets # => "\n"
f.gets # => "Fourth line\n"
f.gets # => "Fifth line\n"
f.gets # => nil
f.close
```

With only string argument `sep` given, returns the next line as determined by line separator `sep`, or `nil` if none; see Line Separator:

```
f = File.new('t.txt')
f.gets('l')   # => "First l"
f.gets('li')  # => "ine\nSecond li"
f.gets('lin') # => "ne\n\nFourth lin"
f.gets        # => "e\n"
f.close
```

The two special values for `sep` are honored:

```
f = File.new('t.txt')
# Get all.
f.gets(nil) # => "First line\nSecond line\n\nFourth line\nFifth line\n"
f.rewind
# Get paragraph (up to two line separators).
f.gets('')  # => "First line\nSecond line\n\n"
f.close
```

With only integer argument `limit` given, limits the number of bytes in the line; see Line Limit:

```
# No more than one line.
File.open('t.txt') {|f| f.gets(10) } # => "First line"
File.open('t.txt') {|f| f.gets(11) } # => "First line\n"
File.open('t.txt') {|f| f.gets(12) } # => "First line\n"
```

With arguments `sep` and `limit` given, combines the two behaviors:

- Returns the next line as determined by line separator `sep`, or `nil` if none.
- But returns no more bytes than are allowed by the limit.

Optional keyword argument `chomp` specifies whether line separators are to be omitted:

```
f = File.open('t.txt')
# Chomp the lines.
f.gets(chomp: true) # => "First line"
f.gets(chomp: true) # => "Second line"
f.gets(chomp: true) # => ""
f.gets(chomp: true) # => "Fourth line"
f.gets(chomp: true) # => "Fifth line"
f.gets(chomp: true) # => nil
f.close
```

`inspect → string`

Returns a string representation of `self`:

```
f = File.open('t.txt')
f.inspect # => "#<File:t.txt>"
f.close
```

`internal_encoding → encoding or nil`

Returns the `Encoding` object that represents the encoding of the internal string, if conversion is specified, or `nil` otherwise.

See Encodings.

ioctl(integer_cmd, argument) → integer

Invokes Posix system call ioctl(2), which issues a low-level command to an I/O device.

Issues a low-level command to an I/O device. The arguments and returned value are platform-dependent. The effect of the call is platform-dependent.

If argument `argument` is an integer, it is passed directly; if it is a string, it is interpreted as a binary sequence of bytes.

Not implemented on all platforms.

isatty → true or false

Returns `true` if the stream is associated with a terminal device (tty), `false` otherwise:

```
f = File.new('t.txt').isatty    #=> false
f.close
f = File.new('/dev/tty').isatty #=> true
f.close
```

Also aliased as: tty?

lineno → integer

Returns the current line number for the stream; see Line Number.

lineno = integer → integer

Sets and returns the line number for the stream; see Line Number.

path → string or nil

Returns the path associated with the `IO`, or `nil` if there is no path associated with the `IO`. It is not guaranteed that the path exists on the filesystem.

```
$stdin.path # => "<STDIN>"

File.open("testfile") {|f| f.path} # => "testfile"
```

Also aliased as: to_path

pid → integer or nil

Returns the process ID of a child process associated with the stream, which will have been set by IO#popen, or `nil` if the stream was not created by IO#popen:

```
pipe = IO.popen("-")
if pipe
  $stderr.puts "In parent, child pid is #{pipe.pid}"
else
  $stderr.puts "In child, pid is #{$$}"
end
```

Output:

```
In child, pid is 26209
In parent, child pid is 26209
```

pos()

Returns the current position (in bytes) in `self` (see Position):

```
f = File.open('t.txt')
f.tell # => 0
f.gets # => "First line\n"
f.tell # => 12
f.close
```

Related: `IO#pos=`, `IO#seek`.

Alias for: tell

pos = new_position → new_position

Seeks to the given `new_position` (in bytes); see Position:

```
f = File.open('t.txt')
f.tell     # => 0
f.pos = 20 # => 20
f.tell     # => 20
f.close
```

Related: `IO#seek`, `IO#tell`.

pread(maxlen, offset) → string
pread(maxlen, offset, out_string) → string

Behaves like `IO#readpartial`, except that it:

- Reads at the given `offset` (in bytes).
- Disregards, and does not modify, the stream's position (see Position).
- Bypasses any user space buffering in the stream.

Because this method does not disturb the stream's state (its position, in particular), `pread` allows multiple threads and processes to use the same IO object for reading at various offsets.

```
f = File.open('t.txt')
f.read # => "First line\nSecond line\n\nFourth line\nFifth line\n"
f.pos  # => 52
# Read 12 bytes at offset 0.
f.pread(12, 0) # => "First line\n"
# Read 9 bytes at offset 8.
f.pread(9, 8)  # => "ne\nSecon"
f.close
```

Not available on some platforms.

`print(*objects) → nil`

Writes the given objects to the stream; returns `nil`. Appends the output record separator `$OUTPUT_RECORD_SEPARATOR` (`$\`), if it is not `nil`. See Line IO.

With argument `objects` given, for each object:

- Converts via its method `to_s` if not a string.
- Writes to the stream.
- If not the last object, writes the output field separator `$OUTPUT_FIELD_SEPARATOR` (`$,`) if it is not `nil`.

With default separators:

```
f = File.open('t.tmp', 'w+')
objects = [0, 0.0, Rational(0, 1), Complex(0, 0), :zero, 'zero']
p $OUTPUT_RECORD_SEPARATOR
p $OUTPUT_FIELD_SEPARATOR
f.print(*objects)
f.rewind
p f.read
f.close
```

Output:

```
nil
nil
"00.00/10+0izerozero"
```

With specified separators:

```
$\ = "\n"
$, = ','
f.rewind
f.print(*objects)
f.rewind
p f.read
```

Output:

```
"0,0.0,0/1,0+0i,zero,zero\n"
```

With no argument given, writes the content of `$_` (which is usually the most recent user input):

```
f = File.open('t.tmp', 'w+')
gets # Sets $_ to the most recent user input.
f.print
f.close
```

`printf(format_string, *objects) → nil`

Formats and writes `objects` to the stream.

For details on `format_string`, see Format Specifications.

`putc(object) → object`

Writes a character to the stream. See Character IO.

If `object` is numeric, converts to integer if necessary, then writes the character whose code is the least significant byte; if `object` is a string, writes the first character:

```
$stdout.putc "A"
$stdout.putc 65
```

Output:

```
AA
```

`puts(*objects) → nil`

Writes the given `objects` to the stream, which must be open for writing; returns `nil`.\ Writes a newline after each that does not already end with a newline sequence. If called without arguments, writes a newline. See Line IO.

Note that each added newline is the character `"\n"</tt>, not the output record separator (<tt>$\`).

Treatment for each object:

- String: writes the string.
- Neither string nor array: writes `object.to_s`.
- Array: writes each element of the array; arrays may be nested.

To keep these examples brief, we define this helper method:

```
def show(*objects)
  # Puts objects to file.
  f = File.new('t.tmp', 'w+')
  f.puts(objects)
  # Return file content.
  f.rewind
  p f.read
  f.close
end

# Strings without newlines.
show('foo', 'bar', 'baz')     # => "foo\nbar\nbaz\n"
# Strings, some with newlines.
show("foo\n", 'bar', "baz\n") # => "foo\nbar\nbaz\n"

# Neither strings nor arrays:
show(0, 0.0, Rational(0, 1), Complex(9, 0), :zero)
# => "0\n0.0\n0/1\n9+0i\nzero\n"

# Array of strings.
show(['foo', "bar\n", 'baz']) # => "foo\nbar\nbaz\n"
# Nested arrays.
show([[[0, 1], 2, 3], 4, 5])  # => "0\n1\n2\n3\n4\n5\n"
```

`pwrite(object, offset) → integer`

Behaves like `IO#write`, except that it:

- Writes at the given `offset` (in bytes).
- Disregards, and does not modify, the stream's position (see Position).
- Bypasses any user space buffering in the stream.

Because this method does not disturb the stream's state (its position, in particular), `pwrite` allows multiple threads and processes to use the same IO object for writing at various offsets.

```
f = File.open('t.tmp', 'w+')
# Write 6 bytes at offset 3.
f.pwrite('ABCDEF', 3) # => 6
f.rewind
```

```
f.read # => "\u0000\u0000\u0000ABCDEF"
f.close
```

Not available on some platforms.

read(maxlen = nil, out_string = nil) → new_string, out_string, or nil

Reads bytes from the stream; the stream must be opened for reading (see Access Modes):

- If `maxlen` is `nil`, reads all bytes using the stream's data mode.
- Otherwise reads up to `maxlen` bytes in binary mode.

Returns a string (either a new string or the given `out_string`) containing the bytes read. The encoding of the string depends on both `maxLen` and `out_string`:

- `maxlen` is `nil`: uses internal encoding of `self` (regardless of whether `out_string` was given).
- `maxlen` not `nil`:
  - `out_string` given: encoding of `out_string` not modified.
  - `out_string` not given: ASCII-8BIT is used.

Without Argument out_string

When argument `out_string` is omitted, the returned value is a new string:

```
f = File.new('t.txt')
f.read
# => "First line\nSecond line\n\nFourth line\nFifth line\n"
f.rewind
f.read(30) # => "First line\r\nSecond line\r\n\r\nFou"
f.read(30) # => "rth line\r\nFifth line\r\n"
f.read(30) # => nil
f.close
```

If `maxlen` is zero, returns an empty string.

With Argument out_string

When argument `out_string` is given, the returned value is `out_string`, whose content is replaced:

```
f = File.new('t.txt')
s = 'foo'      # => "foo"
f.read(nil, s) # => "First line\nSecond line\n\nFourth line\nFifth line\n"
s              # => "First line\nSecond line\n\nFourth line\nFifth line\n"
f.rewind
s = 'bar'
f.read(30, s)  # => "First line\r\nSecond line\r\n\r\nFou"
s              # => "First line\r\nSecond line\r\n\r\nFou"
```

```
s = 'baz'
f.read(30, s)  # => "rth line\r\nFifth line\r\n"
s              # => "rth line\r\nFifth line\r\n"
s = 'bat'
f.read(30, s)  # => nil
s              # => ""
f.close
```

Note that this method behaves like the fread() function in C. This means it retries to invoke read(2) system calls to read data with the specified maxlen (or until EOF).

This behavior is preserved even if the stream is in non-blocking mode. (This method is non-blocking-flag insensitive as other methods.)

If you need the behavior like a single read(2) system call, consider `readpartial`, `read_nonblock`, and `sysread`.

Related: `IO#write`.

read_nonblock(maxlen [, options]) → string
read_nonblock(maxlen, outbuf [, options]) → outbuf

Reads at most *maxlen* bytes from *ios* using the read(2) system call after O_NONBLOCK is set for the underlying file descriptor.

If the optional *outbuf* argument is present, it must reference a `String`, which will receive the data. The *outbuf* will contain only the received data after the method call even if it is not empty at the beginning.

`read_nonblock` just calls the read(2) system call. It causes all errors the read(2) system call causes: Errno::EWOULDBLOCK, Errno::EINTR, etc. The caller should care such errors.

If the exception is Errno::EWOULDBLOCK or Errno::EAGAIN, it is extended by `IO::WaitReadable`. So `IO::WaitReadable` can be used to rescue the exceptions for retrying read_nonblock.

`read_nonblock` causes `EOFError` on EOF.

On some platforms, such as Windows, non-blocking mode is not supported on `IO` objects other than sockets. In such cases, Errno::EBADF will be raised.

If the read byte buffer is not empty, `read_nonblock` reads from the buffer like readpartial. In this case, the read(2) system call is not called.

When `read_nonblock` raises an exception kind of `IO::WaitReadable`, `read_nonblock` should not be called until io is readable for avoiding busy loop. This can be done as follows.

```
# emulates blocking read (readpartial).
begin
  result = io.read_nonblock(maxlen)
rescue IO::WaitReadable
  IO.select([io])
```

```
  retry
end
```

Although IO#read_nonblock doesn't raise IO::WaitWritable. OpenSSL::Buffering#read_nonblock can raise IO::WaitWritable. If IO and SSL should be used polymorphically, IO::WaitWritable should be rescued too. See the document of OpenSSL::Buffering#read_nonblock for sample code.

Note that this method is identical to readpartial except the non-blocking flag is set.

By specifying a keyword argument *exception* to `false`, you can indicate that read_nonblock should not raise an IO::WaitReadable exception, but return the symbol `:wait_readable` instead. At EOF, it will return nil instead of raising EOFError.

readbyte → integer

Reads and returns the next byte (in range 0..255) from the stream; raises EOFError if already at end-of-stream. See Byte IO.

```
f = File.open('t.txt')
f.readbyte # => 70
f.close
f = File.open('t.rus')
f.readbyte # => 209
f.close
```

Related: IO#getbyte (will not raise EOFError).

readchar → string

Reads and returns the next 1-character string from the stream; raises EOFError if already at end-of-stream. See Character IO.

```
f = File.open('t.txt')
f.readchar     # => "F"
f.close
f = File.open('t.rus')
f.readchar.ord # => 1090
f.close
```

Related: IO#getc (will not raise EOFError).

readline(sep = $/, chomp: false) → string
readline(limit, chomp: false) → string
readline(sep, limit, chomp: false) → string

Reads a line as with IO#gets, but raises EOFError if already at end-of-stream.

Optional keyword argument `chomp` specifies whether line separators are to be omitted.

`readlines(sep = $/, chomp: false) → array`
`readlines(limit, chomp: false) → array`
`readlines(sep, limit, chomp: false) → array`

Reads and returns all remaining line from the stream; does not modify `$_`. See Line IO.

With no arguments given, returns lines as determined by line separator `$/`, or `nil` if none:

```
f = File.new('t.txt')
f.readlines
# => ["First line\n", "Second line\n", "\n", "Fourth line\n", "Fifth line\n"]
f.readlines # => []
f.close
```

With only string argument `sep` given, returns lines as determined by line separator `sep`, or `nil` if none; see Line Separator:

```
f = File.new('t.txt')
f.readlines('li')
# => ["First li", "ne\nSecond li", "ne\n\nFourth li", "ne\nFifth li", "ne\n"]
f.close
```

The two special values for `sep` are honored:

```
f = File.new('t.txt')
# Get all into one string.
f.readlines(nil)
# => ["First line\nSecond line\n\nFourth line\nFifth line\n"]
# Get paragraphs (up to two line separators).
f.rewind
f.readlines('')
# => ["First line\nSecond line\n\n", "Fourth line\nFifth line\n"]
f.close
```

With only integer argument `limit` given, limits the number of bytes in each line; see Line Limit:

```
f = File.new('t.txt')
f.readlines(8)
# => ["First li", "ne\n", "Second l", "ine\n", "\n", "Fourth l", "ine\n", "Fi
f.close
```

With arguments `sep` and `limit` given, combines the two behaviors:

- Returns lines as determined by line separator `sep`.
- But returns no more bytes in a line than are allowed by the limit.

Optional keyword argument `chomp` specifies whether line separators are to be omitted:

```
f = File.new('t.txt')
f.readlines(chomp: true)
# => ["First line", "Second line", "", "Fourth line", "Fifth line"]
f.close
```

`readpartial(maxlen) → string`
`readpartial(maxlen, out_string) → out_string`

Reads up to `maxlen` bytes from the stream; returns a string (either a new string or the given `out_string`). Its encoding is:

- The unchanged encoding of `out_string`, if `out_string` is given.
- ASCII-8BIT, otherwise.
- Contains `maxlen` bytes from the stream, if available.
- Otherwise contains all available bytes, if any available.
- Otherwise is an empty string.

With the single non-negative integer argument `maxlen` given, returns a new string:

```
f = File.new('t.txt')
f.readpartial(20) # => "First line\nSecond l"
f.readpartial(20) # => "ine\n\nFourth line\n"
f.readpartial(20) # => "Fifth line\n"
f.readpartial(20) # Raises EOFError.
f.close
```

With both argument `maxlen` and string argument `out_string` given, returns modified `out_string`:

```
f = File.new('t.txt')
s = 'foo'
f.readpartial(20, s) # => "First line\nSecond l"
s = 'bar'
f.readpartial(0, s)  # => ""
f.close
```

This method is useful for a stream such as a pipe, a socket, or a tty. It blocks only when no data is immediately available. This means that it blocks only when *all* of the following are true:

- The byte buffer in the stream is empty.

- The content of the stream is empty.
- The stream is not at EOF.

When blocked, the method waits for either more data or EOF on the stream:

- If more data is read, the method returns the data.
- If EOF is reached, the method raises `EOFError`.

When not blocked, the method responds immediately:

- Returns data from the buffer if there is any.
- Otherwise returns data from the stream if there is any.
- Otherwise raises `EOFError` if the stream has reached EOF.

Note that this method is similar to sysread. The differences are:

- If the byte buffer is not empty, read from the byte buffer instead of “sysread for buffered `IO` (`IOError`)”.
- It doesn’t cause Errno::EWOULDBLOCK and Errno::EINTR. When readpartial meets EWOULDBLOCK and EINTR by read system call, readpartial retries the system call.

The latter means that readpartial is non-blocking-flag insensitive. It blocks on the situation `IO#sysread` causes Errno::EWOULDBLOCK as if the fd is blocking mode.

Examples:

```
#                         # Returned      Buffer Content    Pipe Content
r, w = IO.pipe            #
w << 'abc'                #               ""                "abc".
r.readpartial(4096)       # => "abc"      ""                ""
r.readpartial(4096)       # (Blocks because buffer and pipe are empty.)

#                         # Returned      Buffer Content    Pipe Content
r, w = IO.pipe            #
w << 'abc'                #               ""                "abc"
w.close                   #               ""                "abc" EOF
r.readpartial(4096)       # => "abc"      ""                 EOF
r.readpartial(4096)       # raises EOFError

#                         # Returned      Buffer Content    Pipe Content
r, w = IO.pipe            #
w << "abc\ndef\n"         #               ""                "abc\ndef\n"
r.gets                    # => "abc\n"    "def\n"           ""
w << "ghi\n"              #               "def\n"           "ghi\n"
r.readpartial(4096)       # => "def\n"    ""                "ghi\n"
r.readpartial(4096)       # => "ghi\n"    ""                ""
```

reopen(other_io) → self
reopen(path, mode = 'r', **opts) → self

Reassociates the stream with another stream, which may be of a different class. This method may be used to redirect an existing stream to a new destination.

With argument `other_io` given, reassociates with that stream:

```
# Redirect $stdin from a file.
f = File.open('t.txt')
$stdin.reopen(f)
f.close

# Redirect $stdout to a file.
f = File.open('t.tmp', 'w')
$stdout.reopen(f)
f.close
```

With argument `path` given, reassociates with a new stream to that file path:

```
$stdin.reopen('t.txt')
$stdout.reopen('t.tmp', 'w')
```

Optional keyword arguments `opts` specify:

- Open Options.
- Encoding options.

`rewind → 0`

Repositions the stream to its beginning, setting both the position and the line number to zero; see Position and Line Number:

```
f = File.open('t.txt')
f.tell     # => 0
f.lineno   # => 0
f.gets     # => "First line\n"
f.tell     # => 12
f.lineno   # => 1
f.rewind   # => 0
f.tell     # => 0
f.lineno   # => 0
f.close
```

Note that this method cannot be used with streams such as pipes, ttys, and sockets.

`seek(offset, whence = IO::SEEK_SET) → 0`

Seeks to the position given by integer `offset` (see Position) and constant `whence`, which is one of:

- `:CUR` or `IO::SEEK_CUR`: Repositions the stream to its current position plus the given `offset`:

```
f = File.open('t.txt')
f.tell            # => 0
f.seek(20, :CUR)  # => 0
f.tell            # => 20
f.seek(-10, :CUR) # => 0
f.tell            # => 10
f.close
```

- `:END` or `IO::SEEK_END`: Repositions the stream to its end plus the given `offset`:

```
f = File.open('t.txt')
f.tell            # => 0
f.seek(0, :END)   # => 0  # Repositions to stream end.
f.tell            # => 52
f.seek(-20, :END) # => 0
f.tell            # => 32
f.seek(-40, :END) # => 0
f.tell            # => 12
f.close
```

- `:SET` or `IO:SEEK_SET`: Repositions the stream to the given `offset`:

```
f = File.open('t.txt')
f.tell            # => 0
f.seek(20, :SET) # => 0
f.tell           # => 20
f.seek(40, :SET) # => 0
f.tell           # => 40
f.close
```

Related: `IO#pos=`, `IO#tell`.

`set_encoding(ext_enc) → self`
`set_encoding(ext_enc, int_enc, **enc_opts) → self`
`set_encoding('ext_enc:int_enc', **enc_opts) → self`

See Encodings.

Argument `ext_enc`, if given, must be an `Encoding` object or a `String` with the encoding name; it is assigned as the encoding for the stream.

Argument `int_enc`, if given, must be an `Encoding` object or a `String` with the encoding name; it is assigned as the encoding for the internal string.

Argument `'ext_enc:int_enc'`, if given, is a string containing two colon-separated encoding names; corresponding `Encoding` objects are assigned as the external and internal encodings for the stream.

If the external encoding of a string is binary/ASCII-8BIT, the internal encoding of the string is set to nil, since no transcoding is needed.

Optional keyword arguments `enc_opts` specify Encoding options.

set_encoding_by_bom → encoding or nil

If the stream begins with a BOM (byte order marker), consumes the BOM and sets the external encoding accordingly; returns the result encoding if found, or `nil` otherwise:

```
File.write('t.tmp', "\u{FEFF}abc")
io = File.open('t.tmp', 'rb')
io.set_encoding_by_bom # => #<Encoding:UTF-8>
io.close

File.write('t.tmp', 'abc')
io = File.open('t.tmp', 'rb')
io.set_encoding_by_bom # => nil
io.close
```

Raises an exception if the stream is not binmode or its encoding has already been set.

stat → stat

Returns status information for *ios* as an object of type `File::Stat`.

```
f = File.new("testfile")
s = f.stat
"%o" % s.mode   #=> "100644"
s.blksize       #=> 4096
s.atime         #=> Wed Apr 09 08:53:54 CDT 2003
```

sync → true or false

Returns the current sync mode of the stream. When sync mode is true, all output is immediately flushed to the underlying operating system and is not buffered by Ruby internally. See also `fsync`.

```
f = File.open('t.tmp', 'w')
f.sync # => false
f.sync = true
f.sync # => true
f.close
```

sync = boolean → boolean

Sets the *sync mode* for the stream to the given value; returns the given value.

Values for the sync mode:

- `true`: All output is immediately flushed to the underlying operating system and is not buffered internally.
- `false`: Output may be buffered internally.

Example;

```
f = File.open('t.tmp', 'w')
f.sync # => false
f.sync = true
f.sync # => true
f.close
```

Related: IO#fsync.

sysread(maxlen) → string
sysread(maxlen, out_string) → string

Behaves like IO#readpartial, except that it uses low-level system functions.

This method should not be used with other stream-reader methods.

sysseek(offset, whence = IO::SEEK_SET) → integer

Behaves like IO#seek, except that it:

- Uses low-level system functions.
- Returns the new position.

syswrite(object) → integer

Writes the given `object` to self, which must be opened for writing (see Modes); returns the number bytes written. If `object` is not a string is converted via method to_s:

```
f = File.new('t.tmp', 'w')
f.syswrite('foo') # => 3
f.syswrite(30)    # => 2
f.syswrite(:foo)  # => 3
f.close
```

This methods should not be used with other stream-writer methods.

tell → integer

Returns the current position (in bytes) in `self` (see Position):

```
f = File.open('t.txt')
f.tell # => 0
f.gets # => "First line\n"
```

```
f.tell # => 12
f.close
```

Related: IO#pos=, IO#seek.

Also aliased as: pos

timeout → duration or nil

Get the internal timeout duration or nil if it was not set.

timeout = duration → duration
timeout = nil → nil

Sets the internal timeout to the specified duration or nil. The timeout applies to all blocking operations where possible.

When the operation performs longer than the timeout set, IO::TimeoutError is raised.

This affects the following methods (but is not limited to): gets, puts, read, write, wait_readable and wait_writable. This also affects blocking socket operations like Socket#accept and Socket#connect.

Some operations like File#open and IO#close are not affected by the timeout. A timeout during a write operation may leave the IO in an inconsistent state, e.g. data was partially written. Generally speaking, a timeout is a last ditch effort to prevent an application from hanging on slow I/O operations, such as those that occur during a slowloris attack.

to_i()

Returns the integer file descriptor for the stream:

```
$stdin.fileno             # => 0
$stdout.fileno            # => 1
$stderr.fileno            # => 2
File.open('t.txt').fileno # => 10
f.close
```

Alias for: fileno

to_io → self

Returns `self`.

to_path()

Returns the path associated with the `IO`, or `nil` if there is no path associated with the `IO`. It is not guaranteed that the path exists on the filesystem.

```
$stdin.path # => "<STDIN>"

File.open("testfile") {|f| f.path} # => "testfile"
```

Alias for: path

tty?()

Returns `true` if the stream is associated with a terminal device (tty), `false` otherwise:

```
f = File.new('t.txt').isatty    #=> false
f.close
f = File.new('/dev/tty').isatty #=> true
f.close
```

Alias for: isatty

ungetbyte(integer) → nil
ungetbyte(string) → nil

Pushes back (“unshifts”) the given data onto the stream’s buffer, placing the data so that it is next to be read; returns `nil`. See Byte IO.

Note that:

- Calling the method has no effect with unbuffered reads (such as `IO#sysread`).
- Calling `rewind` on the stream discards the pushed-back data.

When argument `integer` is given, uses only its low-order byte:

```
File.write('t.tmp', '012')
f = File.open('t.tmp')
f.ungetbyte(0x41)   # => nil
f.read              # => "A012"
f.rewind
f.ungetbyte(0x4243) # => nil
f.read              # => "C012"
f.close
```

When argument `string` is given, uses all bytes:

```
File.write('t.tmp', '012')
f = File.open('t.tmp')
f.ungetbyte('A')    # => nil
f.read              # => "A012"
f.rewind
f.ungetbyte('BCDE') # => nil
f.read              # => "BCDE012"
f.close
```

ungetc(integer) → nil
ungetc(string) → nil

Pushes back (“unshifts”) the given data onto the stream’s buffer, placing the data so that it is next to be read; returns `nil`. See Character IO.

Note that:

- Calling the method has no effect with unbuffered reads (such as `IO#sysread`).
- Calling `rewind` on the stream discards the pushed-back data.

When argument `integer` is given, interprets the integer as a character:

```
File.write('t.tmp', '012')
f = File.open('t.tmp')
f.ungetc(0x41)     # => nil
f.read             # => "A012"
f.rewind
f.ungetc(0x0442)   # => nil
f.getc.ord         # => 1090
f.close
```

When argument `string` is given, uses all characters:

```
File.write('t.tmp', '012')
f = File.open('t.tmp')
f.ungetc('A')      # => nil
f.read      # => "A012"
f.rewind
f.ungetc("\u0442\u0435\u0441\u0442") # => nil
f.getc.ord      # => 1090
f.getc.ord      # => 1077
f.getc.ord      # => 1089
f.getc.ord      # => 1090
f.close
```

wait(events, timeout) → event mask, false or nil
wait(timeout = nil, mode = :read) → self, true, or false

Waits until the IO becomes ready for the specified events and returns the subset of events that become ready, or a falsy value when times out.

The events can be a bit mask of `IO::READABLE`, `IO::WRITABLE` or `IO::PRIORITY`.

Returns an event mask (truthy value) immediately when buffered data is available.

Optional parameter `mode` is one of `:read`, `:write`, or `:read_write`.

wait_priority → truthy or falsy
wait_priority(timeout) → truthy or falsy

Waits until IO is priority and returns a truthy value or a falsy value when times out. Priority data is sent and received using the Socket::MSG_OOB flag and is typically limited to streams.

wait_readable → truthy or falsy
wait_readable(timeout) → truthy or falsy

Waits until IO is readable and returns a truthy value, or a falsy value when times out. Returns a truthy value immediately when buffered data is available.

wait_writable → truthy or falsy
wait_writable(timeout) → truthy or falsy

Waits until IO is writable and returns a truthy value or a falsy value when times out.

write(*objects) → integer

Writes each of the given `objects` to `self`, which must be opened for writing (see Access Modes); returns the total number bytes written; each of `objects` that is not a string is converted via method `to_s`:

```
$stdout.write('Hello', ', ', 'World!', "\n") # => 14
$stdout.write('foo', :bar, 2, "\n")          # => 8
```

Output:

```
Hello, World!
foobar2
```

Related: IO#read.

write_nonblock(string) → integer
write_nonblock(string [, options]) → integer

Writes the given string to *ios* using the write(2) system call after O_NONBLOCK is set for the underlying file descriptor.

It returns the number of bytes written.

write_nonblock just calls the write(2) system call. It causes all errors the write(2) system call causes: Errno::EWOULDBLOCK, Errno::EINTR, etc. The result may also be smaller than string.length (partial write). The caller should care such errors and partial write.

If the exception is Errno::EWOULDBLOCK or Errno::EAGAIN, it is extended by IO::WaitWritable. So IO::WaitWritable can be used to rescue the exceptions for retrying write_nonblock.

```
# Creates a pipe.
r, w = IO.pipe

# write_nonblock writes only 65536 bytes and return 65536.
# (The pipe size is 65536 bytes on this environment.)
s = "a" * 100000
p w.write_nonblock(s)     #=> 65536

# write_nonblock cannot write a byte and raise EWOULDBLOCK (EAGAIN).
p w.write_nonblock("b")   # Resource temporarily unavailable (Errno::EAGAIN)
```

If the write buffer is not empty, it is flushed at first.

When write_nonblock raises an exception kind of IO::WaitWritable, write_nonblock should not be called until io is writable for avoiding busy loop. This can be done as follows.

```
begin
  result = io.write_nonblock(string)
rescue IO::WaitWritable, Errno::EINTR
  IO.select(nil, [io])
  retry
end
```

Note that this doesn't guarantee to write all data in string. The length written is reported as result and it should be checked later.

On some platforms such as Windows, write_nonblock is not supported according to the kind of the IO object. In such cases, write_nonblock raises Errno::EBADF.

By specifying a keyword argument *exception* to `false`, you can indicate that write_nonblock should not raise an IO::WaitWritable exception, but return the symbol `:wait_writable` instead.

Validate

Generated by RDoc 6.4.0.

Based on Darkfish by Michael Granger.

Ruby-doc.org is a service of James Britt and Neurogami, purveyors of fine dance noise