NAME

"IO::Async::Stream" – event callbacks and write bufering for a stream filehandle

SYNOPSIS

```
use IO::Async::Stream;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

my $stream = IO::Async::Stream->new(
   read_handle  => \*STDIN,
   write_handle => \*STDOUT,

   on_read => sub {
      my ( $self, $buffref, $eof ) = @_;

      while( $$buffref =~ s/^(.*\n)// ) {
         print "Received a line $1";
      }

      if( $eof ) {
         print "EOF; last partial line is $$buffref\n";
      }

      return 0;
   }
);

$loop->add( $stream );

$stream->write( "An initial line here\n" );
```

DESCRIPTION

This subclass of IO::Async::Handle contains a filehandle that represents a byte-stream. It provides buffering for both incoming and outgoing data. It invokes the `on_read` handler when new data is read from the filehandle. Data may be written to the filehandle by calling the `write` method.

This class is suitable for any kind of filehandle that provides a possibly-bidirectional reliable byte stream, such as a pipe, TTY, or `SOCK_STREAM` socket (such as TCP or a byte-oriented UNIX local socket). For datagram or raw message-based sockets (such as UDP) see instead IO::Async::Socket.

EVENTS

The following events are invoked, either using subclass methods or CODE references in parameters:

`$ret` **= on_read \$buffer,** `$eof`

Invoked when more data is available in the internal receiving buffer.

The first argument is a reference to a plain perl string. The code should inspect and remove any data it likes, but is not required to remove all, or indeed any of the data. Any data remaining in the buffer will be preserved for the next call, the next time more data is received from the handle.

In this way, it is easy to implement code that reads records of some form when completed, but ignores partially-received records, until all the data is present. If the handler wishes to be immediately invoke a second time, to have another attempt at consuming more content, it should return `1`. Otherwise, it should return `0`, and the handler will next be invoked when more data has arrived from the underlying read handle and appended to the buffer. This makes it easy to implement code that handles multiple incoming records at the same time. Alternatively, if the handler function already attempts to consume as much as possible from the buffer, it will have no need to return `1` at all. See the examples at the end of this documentation for more detail.

The second argument is a scalar indicating whether the stream has reported an end-of-file (EOF) condition. A reference to the buffer is passed to the handler in the usual way, so it may inspect data contained in it. Once the handler returns a false value, it will not be called again, as the handle is now at EOF and no more data can arrive.

The `on_read` code may also dynamically replace itself with a new callback by returning a CODE reference instead of `0` or `1`. The original callback or method that the object first started with may be restored by returning `undef`. Whenever the callback is changed in this way, the new code is called again; even if the read buffer is currently empty. See the examples at the end of this documentation for more detail.

The `push_on_read` method can be used to insert new, temporary handlers that take precedence over the global `on_read` handler. This event is only used if there are no further pending handlers created by `push_on_read`.

on_read_eof
Optional. Invoked when the read handle indicates an end-of-file (EOF) condition. If there is any data in the buffer still to be processed, the `on_read` event will be invoked first, before this one.

on_write_eof
Optional. Invoked when the write handle indicates an end-of-file (EOF) condition. Note that this condition can only be detected after a `write` syscall returns the `EPIPE` error. If there is no data pending to be written then it will not be detected yet.

on_read_error `$errno`
Optional. Invoked when the `sysread` method on the read handle fails.

on_write_error `$errno`
Optional. Invoked when the `syswrite` method on the write handle fails.

The `on_read_error` and `on_write_error` handlers are passed the value of `$!` at the time the error occurred. (The `$!` variable itself, by its nature, may have changed from the original error by the time this handler runs so it should always use the value passed in).

If an error occurs when the corresponding error callback is not supplied, and there is not a handler for it, then the `close` method is called instead.

on_read_high_watermark `$length`
on_read_low_watermark `$length`
Optional. Invoked when the read buffer grows larger than the high watermark or smaller than the low watermark respectively. These are edge-triggered events; they will only be triggered once per crossing, not continuously while the buffer remains above or below the given limit.

If these event handlers are not defined, the default behaviour is to disable read-ready notifications if the read buffer grows larger than the high watermark (so as to avoid it growing arbitrarily if nothing is consuming it), and re-enable notifications again once something has read enough to cause it to drop. If these events are overridden, the overriding code will have to perform this behaviour if required, by using

```
$self->want_readready_for_read(...)
```

on_outgoing_empty
Optional. Invoked when the writing data buffer becomes empty.

on_writeable_start
on_writeable_stop
Optional. These two events inform when the filehandle becomes writeable, and when it stops being writeable. `on_writeable_start` is invoked by the `on_write_ready` event if previously it was known to be not writeable. `on_writeable_stop` is invoked after a `syswrite` operation fails with `EAGAIN` or `EWOULDBLOCK`. These two events track the writeability state, and ensure that only state change cause events to be invoked. A stream starts off being presumed writeable, so the first of these events to be observed will be `on_writeable_stop`.

PARAMETERS

The following named parameters may be passed to `new` or `configure`:

read_handle => IO
The IO handle to read from. Must implement `fileno` and `sysread` methods.

write_handle => IO
The IO handle to write to. Must implement `fileno` and `syswrite` methods.

handle => IO
Shortcut to specifying the same IO handle for both of the above.

on_read => CODE
on_read_error => CODE
on_outgoing_empty => CODE
on_write_error => CODE
on_writeable_start => CODE
on_writeable_stop => CODE
CODE references for event handlers.

autoflush => BOOL
Optional. If true, the `write` method will attempt to write data to the operating system immediately, without waiting for the loop to indicate the filehandle is write-ready. This is useful, for example, on streams that should contain up-to-date logging or console information.

It currently defaults to false for any file handle, but future versions of IO::Async may enable this by default on STDOUT and STDERR.

read_len => INT
Optional. Sets the buffer size for `read` calls. Defaults to 8 KiBytes.

read_all => BOOL
Optional. If true, attempt to read as much data from the kernel as possible when the handle becomes readable. By default this is turned off, meaning at most one fixed-size buffer is read. If there is still more data in the kernel's buffer, the handle will still be readable, and will be read from again.

This behaviour allows multiple streams and sockets to be multiplexed simultaneously, meaning that a large bulk transfer on one cannot starve other filehandles of processing time. Turning this option on may improve bulk data transfer rate, at the risk of delaying or stalling processing on other filehandles.

write_len => INT
Optional. Sets the buffer size for `write` calls. Defaults to 8 KiBytes.

write_all => BOOL
Optional. Analogous to the `read_all` option, but for writing. When `autoflush` is enabled, this option only affects deferred writing if the initial attempt failed due to buffer space.

read_high_watermark => INT
read_low_watermark => INT
Optional. If defined, gives a way to implement flow control or other behaviours that depend on the size of Stream's read buffer.

If after more data is read from the underlying filehandle the read buffer is now larger than the high watermark, the `on_read_high_watermark` event is triggered (which, by default, will disable read-ready notifications and pause reading from the filehandle).

If after data is consumed by an `on_read` handler the read buffer is now smaller than the low watermark, the `on_read_low_watermark` event is triggered (which, by default, will re-enable read-ready notifications and resume reading from the filehandle). For to be possible, the read handler would have to be one added by the `push_on_read` method or one of the Future-returning `read_*` methods.

By default these options are not defined, so this behaviour will not happen. `read_low_watermark` may not be set to a larger value than `read_high_watermark`, but it may be set to a smaller value, creating a

hysteresis region. If either option is defined then both must be.

If these options are used with the default event handlers, be careful not to cause deadlocks by having a high watermark sufficiently low that a single `on_read` invocation might not consider it finished yet.

reader => STRING|CODE
writer => STRING|CODE

Optional. If defined, gives the name of a method or a CODE reference to use to implement the actual reading from or writing to the filehandle. These will be invoked as

```
$stream->reader( $read_handle, $buffer, $len )
$stream->writer( $write_handle, $buffer, $len )
```

Each is expected to modify the passed buffer; `reader` by appending to it, `writer` by removing a prefix from it. Each is expected to return a true value on success, zero on EOF, or `undef` with `$!` set for errors. If not provided, they will be substituted by implenentations using `sysread` and `syswrite` on the underlying handle, respectively.

close_on_read_eof => BOOL

Optional. Usually true, but if set to a false value then the stream will not be `close`d when an EOF condition occurs on read. This is normally not useful as at that point the underlying stream filehandle is no longer useable, but it may be useful for reading regular files, or interacting with TTY devices.

encoding => STRING

If supplied, sets the name of encoding of the underlying stream. If an encoding is set, then the `write` method will expect to receive Unicode strings and encodes them into bytes, and incoming bytes will be decoded into Unicode strings for the `on_read` event.

If an encoding is not supplied then `write` and `on_read` will work in byte strings.

IMPORTANT NOTE: in order to handle reads of UTF-8 content or other multibyte encodings, the code implementing the `on_read` event uses a feature of Encode; the `STOP_AT_PARTIAL` flag. While this flag has existed for a while and is used by the `:encoding` PerlIO layer itself for similar purposes, the flag is not officially documented by the `Encode` module. In principle this undocumented feature could be subject to change, in practice I believe it to be reasonably stable.

This note applies only to the `on_read` event; data written using the `write` method does not rely on any undocumented features of `Encode`.

If a read handle is given, it is required that either an `on_read` callback reference is configured, or that the object provides an `on_read` method. It is optional whether either is true for `on_outgoing_empty`; if neither is supplied then no action will be taken when the writing buffer becomes empty.

An `on_read` handler may be supplied even if no read handle is yet given, to be used when a read handle is eventually provided by the `set_handles` method.

This condition is checked at the time the object is added to a Loop; it is allowed to create a `IO::Async::Stream` object with a read handle but without a `on_read` handler, provided that one is later given using `configure` before the stream is added to its containing Loop, either directly or by being a child of another Notifier already in a Loop, or added to one.

METHODS

The following methods documented with a trailing call to `->get` return Future instances.

want_readready_for_read
want_readready_for_write

```
$stream->want_readready_for_read( $set )

$stream->want_readready_for_write( $set )
```

Mutators for the `want_readready` property on IO::Async::Handle, which control whether the `read` or `write` behaviour should be continued once the filehandle becomes ready for read.

Normally, `want_readready_for_read` is always true (though the read watermark behaviour can

modify it), and `want_readready_for_write` is not used. However, if a custom `writer` function is provided, it may find this useful for being invoked again if it cannot proceed with a write operation until the filehandle becomes readable (such as during transport negotiation or SSL key management, for example).

want_writeready_for_read
want_writeready_for_write

```
$stream->want_writeready_for_write( $set )

$stream->want_writeready_for_read( $set )
```

Mutators for the `want_writeready` property on IO::Async::Handle, which control whether the `write` or `read` behaviour should be continued once the filehandle becomes ready for write.

Normally, `want_writeready_for_write` is managed by the `write` method and associated flushing, and `want_writeready_for_read` is not used. However, if a custom `reader` function is provided, it may find this useful for being invoked again if it cannot proceed with a read operation until the filehandle becomes writable (such as during transport negotiation or SSL key management, for example).

close

```
$stream->close
```

A synonym for `close_when_empty`. This should not be used when the deferred wait behaviour is required, as the behaviour of `close` may change in a future version of IO::Async. Instead, call `close_when_empty` directly.

close_when_empty

```
$stream->close_when_empty
```

If the write buffer is empty, this method calls `close` on the underlying IO handles, and removes the stream from its containing loop. If the write buffer still contains data, then this is deferred until the buffer is empty. This is intended for "write-then-close" one-shot streams.

```
$stream->write( "Here is my final data\n" );
$stream->close_when_empty;
```

Because of this deferred nature, it may not be suitable for error handling. See instead the `close_now` method.

close_now

```
$stream->close_now
```

This method immediately closes the underlying IO handles and removes the stream from the containing loop. It will not wait to flush the remaining data in the write buffer.

is_read_eof
is_write_eof

```
$eof = $stream->is_read_eof

$eof = $stream->is_write_eof
```

Returns true after an EOF condition is reported on either the read or the write handle, respectively.

write

```
$stream->write( $data, %params )
```

This method adds data to the outgoing data queue, or writes it immediately, according to the `autoflush` parameter.

If the `autoflush` option is set, this method will try immediately to write the data to the underlying filehandle. If this completes successfully then it will have been written by the time this method returns. If it fails to write completely, then the data is queued as if `autoflush` were not set, and will be flushed as normal.

`$data` can either be a plain string, a Future, or a CODE reference. If it is a plain string it is written immediately. If it is not, its value will be used to generate more `$data` values, eventually leading to strings

to be written.

If `$data` is a `Future`, the Stream will wait until it is ready, and take the single value it yields.

If `$data` is a CODE reference, it will be repeatedly invoked to generate new values. Each time the filehandle is ready to write more data to it, the function is invoked. Once the function has finished generating data it should return undef. The function is passed the Stream object as its first argument.

It is allowed that `Futures` yield CODE references, or CODE references return `Futures`, as well as plain strings.

For example, to stream the contents of an existing opened filehandle:

```
open my $fileh, "<", $path or die "Cannot open $path - $!";

$stream->write( sub {
   my ( $stream ) = @_;

   sysread $fileh, my $buffer, 8192 or return;
   return $buffer;
} );
```

Takes the following optional named parameters in `%params`:

write_len => INT
: Overrides the `write_len` parameter for the data written by this call.

on_write => CODE
: A CODE reference which will be invoked after every successful `syswrite` operation on the underlying filehandle. It will be passed the number of bytes that were written by this call, which may not be the entire length of the buffer – if it takes more than one `syscall` operation to empty the buffer then this callback will be invoked multiple times.

```
$on_write->( $stream, $len )
```

on_flush => CODE
: A CODE reference which will be invoked once the data queued by this `write` call has been flushed. This will be invoked even if the buffer itself is not yet empty; if more data has been queued since the call.

```
$on_flush->( $stream )
```

on_error => CODE
: A CODE reference which will be invoked if a `syswrite` error happens while performing this write. Invoked as for the `Stream`'s `on_write_error` event.

```
$on_error->( $stream, $errno )
```

If the object is not yet a member of a loop and doesn't yet have a `write_handle`, then calls to the `write` method will simply queue the data and return. It will be flushed when the object is added to the loop.

If `$data` is a defined but empty string, the write is still queued, and the `on_flush` continuation will be invoked, if supplied. This can be used to obtain a marker, to invoke some code once the output queue has been flushed up to this point.

write (scalar)

```
$stream->write( ... )->get
```

If called in non-void context, this method returns a Future which will complete (with no value) when the write operation has been flushed. This may be used as an alternative to, or combined with, the `on_flush` callback.

push_on_read

```
$stream->push_on_read( $on_read )
```

Pushes a new temporary `on_read` handler to the end of the queue. This queue, if non-empty, is used to provide `on_read` event handling code in preference to using the object's main event handler or method. New handlers can be supplied at any time, and they will be used in first-in first-out (FIFO) order.

As with the main `on_read` event handler, each can return a (defined) boolean to indicate if they wish to be invoked again or not, another `CODE` reference to replace themself with, or `undef` to indicate it is now complete and should be removed. When a temporary handler returns `undef` it is shifted from the queue and the next one, if present, is invoked instead. If there are no more then the object's main handler is invoked instead.

FUTURE-RETURNING READ METHODS

The following methods all return a Future which will become ready when enough data has been read by the Stream into its buffer. At this point, the data is removed from the buffer and given to the `Future` object to complete it.

```
my $f = $stream->read_...

my ( $string ) = $f->get;
```

Unlike the `on_read` event handlers, these methods don't allow for access to "partial" results; they only provide the final result once it is ready.

If a `Future` is cancelled before it completes it is removed from the read queue without consuming any data; i.e. each `Future` atomically either completes or is cancelled.

Since it is possible to use a readable `Stream` entirely using these `Future`-returning methods instead of the `on_read` event, it may be useful to configure a trivial return-false event handler to keep it from consuming any input, and to allow it to be added to a `Loop` in the first place.

```
my $stream = IO::Async::Stream->new( on_read => sub { 0 }, ... );
$loop->add( $stream );

my $f = $stream->read_...
```

If a read EOF or error condition happens while there are read `Future`s pending, they are all completed. In the case of a read EOF, they are done with `undef`; in the case of a read error they are failed using the `$!` error value as the failure.

```
$f->fail( $message, sysread => $! )
```

If a read EOF condition happens to the currently-processing read `Future`, it will return a partial result. The calling code can detect this by the fact that the returned data is not complete according to the specification (too short in `read_exactly`'s case, or lacking the ending pattern in `read_until`'s case). Additionally, each `Future` will yield the `$eof` value in its results.

```
my ( $string, $eof ) = $f->get;
```

read_atmost

read_exactly

```
( $string, $eof ) = $stream->read_atmost( $len )->get

( $string, $eof ) = $stream->read_exactly( $len )->get
```

Completes the `Future` when the read buffer contains `$len` or more characters of input. `read_atmost` will also complete after the first invocation of `on_read`, even if fewer characters are available, whereas `read_exactly` will wait until at least `$len` are available.

read_until

```
( $string, $eof ) = $stream->read_until( $end )->get
```

Completes the `Future` when the read buffer contains a match for `$end`, which may either be a plain

string or a compiled `Regexp` reference. Yields the prefix of the buffer up to and including this match.

read_until_eof

```
( $string, $eof ) = $stream->read_until_eof->get
```

Completes the `Future` when the stream is eventually closed at EOF, and yields all of the data that was available.

UTILITY CONSTRUCTORS

new_for_stdin
new_for_stdout
new_for_stdio

```
$stream = IO::Async::Stream->new_for_stdin

$stream = IO::Async::Stream->new_for_stdout

$stream = IO::Async::Stream->new_for_stdio
```

Return a `IO::Async::Stream` object preconfigured with the correct `read_handle`, `write_handle` or both.

connect

```
$future = $stream->connect( %args )
```

A convenient wrapper for calling the `connect` method on the underlying IO::Async::Loop object, passing the `socktype` hint as `stream` if not otherwise supplied.

DEBUGGING FLAGS

The following flags in `IO_ASYNC_DEBUG_FLAGS` enable extra logging:

`Sr` Log byte buffers as data is read from a Stream

`Sw` Log byte buffers as data is written to a Stream

EXAMPLES

A line-based `on_read` method

The following `on_read` method accepts incoming `\n`–terminated lines and prints them to the program's `STDOUT` stream.

```
sub on_read
{
   my $self = shift;
   my ( $buffref, $eof ) = @_;

   while( $$buffref =~ s/^(.*\n)// ) {
      print "Received a line: $1";
   }

   return 0;
}
```

Because a reference to the buffer itself is passed, it is simple to use a `s///` regular expression on the scalar it points at, to both check if data is ready (i.e. a whole line), and to remove it from the buffer. Since it always removes as many complete lines as possible, it doesn't need invoking again when it has finished, so it can return a constant `0`.

Reading binary data

This `on_read` method accepts incoming records in 16–byte chunks, printing each one.

```
sub on_read
{
   my ( $self, $buffref, $eof ) = @_;
```

```
   if( length $$buffref >= 16 ) {
      my $record = substr( $$buffref, 0, 16, "" );
      print "Received a 16-byte record: $record\n";

      return 1;
   }

   if( $eof and length $$buffref ) {
      print "EOF: a partial record still exists\n";
   }

   return 0;
}
```

This time, rather than a `while()` loop we have decided to have the handler just process one record, and use the `return 1` mechanism to ask that the handler be invoked again if there still remains data that might contain another record; only stopping with `return 0` when we know we can't find one.

The 4–argument form of `substr()` extracts the 16–byte record from the buffer and assigns it to the `$record` variable, if there was enough data in the buffer to extract it.

A lot of protocols use a fixed-size header, followed by a variable-sized body of data, whose size is given by one of the fields of the header. The following `on_read` method extracts messages in such a protocol.

```
sub on_read
{
   my ( $self, $buffref, $eof ) = @_;

   return 0 unless length $$buffref >= 8; # "N n n" consumes 8 bytes

   my ( $len, $x, $y ) = unpack "N n n", $$buffref;

   return 0 unless length $$buffref >= 8 + $len;

   substr( $$buffref, 0, 8, "" );
   my $data = substr( $$buffref, 0, $len, "" );

   print "A record with values x=$x y=$y\n";

   return 1;
}
```

In this example, the header is `unpack()`ed first, to extract the body length, and then the body is extracted. If the buffer does not have enough data yet for a complete message then `0` is returned, and the buffer is left unmodified for next time. Only when there are enough bytes in total does it use `substr()` to remove them.

Dynamic replacement of `on_read`

Consider the following protocol (inspired by IMAP), which consists of `\n`–terminated lines that may have an optional data block attached. The presence of such a data block, as well as its size, is indicated by the line prefix.

```
sub on_read
{
   my $self = shift;
   my ( $buffref, $eof ) = @_;

   if( $$buffref =~ s/^DATA (\d+):(.*)\n// ) {
      my $length = $1;
```

```
            my $line   = $2;

            return sub {
               my $self = shift;
               my ( $buffref, $eof ) = @_;

               return 0 unless length $$buffref >= $length;

               # Take and remove the data from the buffer
               my $data = substr( $$buffref, 0, $length, "" );

               print "Received a line $line with some data ($data)\n";

               return undef; # Restore the original method
            }
         }
         elsif( $$buffref =~ s/^LINE:(.*)\n// ) {
            my $line = $1;

            print "Received a line $line with no data\n";

            return 1;
         }
         else {
            print STDERR "Unrecognised input\n";
            # Handle it somehow
         }
      }
```

In the case where trailing data is supplied, a new temporary `on_read` callback is provided in a closure. This closure captures the `$length` variable so it knows how much data to expect. It also captures the `$line` variable so it can use it in the event report. When this method has finished reading the data, it reports the event, then restores the original method by returning `undef`.

SEE ALSO

- IO::Handle – Supply object methods for I/O handles

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>