## NAME

AnyEvent::Handle − non−blocking I/O on streaming handles via AnyEvent

## SYNOPSIS

```
use AnyEvent;
use AnyEvent::Handle;

my $cv = AnyEvent->condvar;

my $hdl; $hdl = new AnyEvent::Handle
   fh => \*STDIN,
   on_error => sub {
      my ($hdl, $fatal, $msg) = @_;
      AE::log error => $msg;
      $hdl->destroy;
      $cv->send;
   };

# send some request line
$hdl->push_write ("getinfo\015\012");

# read the response line
$hdl->push_read (line => sub {
   my ($hdl, $line) = @_;
   say "got line <$line>";
   $cv->send;
});

$cv->recv;
```

## DESCRIPTION

This is a helper module to make it easier to do event-based I/O on stream-based filehandles (sockets, pipes, and other stream things). Specifically, it doesn't work as expected on files, packet-based sockets or similar things.

The AnyEvent::Intro tutorial contains some well-documented AnyEvent::Handle examples.

In the following, where the documentation refers to "bytes", it means characters. As sysread and syswrite are used for all I/O, their treatment of characters applies to this module as well.

At the very minimum, you should specify `fh` or `connect`, and the `on_error` callback.

All callbacks will be invoked with the handle object as their first argument.

## METHODS

`$handle` = **new** AnyEvent::Handle fh => `$filehandle`, key => value...

The constructor supports these arguments (all as `key => value` pairs).

fh => `$filehandle` [`fh` or `connect` MANDATORY]

The filehandle this AnyEvent::Handle object will operate on. NOTE: The filehandle will be set to non-blocking mode (using `AnyEvent::fh_unblock`) by the constructor and needs to stay in that mode.

connect => [$host, `$service`] [`fh` or `connect` MANDATORY]

Try to connect to the specified host and service (port), using `AnyEvent::Socket::tcp_connect`. The `$host` additionally becomes the default `peername`.

You have to specify either this parameter, or `fh`, above.

It is possible to push requests on the read and write queues, and modify properties of the stream,

even while AnyEvent::Handle is connecting.

When this parameter is specified, then the `on_prepare`, `on_connect_error` and `on_connect` callbacks will be called under the appropriate circumstances:

on_prepare => `$cb->($handle)`
: This (rarely used) callback is called before a new connection is attempted, but after the file handle has been created (you can access that file handle via `$handle->{fh}`). It could be used to prepare the file handle with parameters required for the actual connect (as opposed to settings that can be changed when the connection is already established).

  The return value of this callback should be the connect timeout value in seconds (or `0`, or `undef`, or the empty list, to indicate that the default timeout is to be used).

on_connect => `$cb->($handle, $host, $port, $retry->())`
: This callback is called when a connection has been successfully established.

  The peer's numeric host and port (the socket peername) are passed as parameters, together with a retry callback. At the time it is called the read and write queues, EOF status, TLS status and similar properties of the handle will have been reset.

  If, for some reason, the handle is not acceptable, calling `$retry` will continue with the next connection target (in case of multi-homed hosts or SRV records there can be multiple connection endpoints). The `$retry` callback can be invoked after the connect callback returns, i.e. one can start a handshake and then decide to retry with the next host if the handshake fails.

  In most cases, you should ignore the `$retry` parameter.

on_connect_error => `$cb->($handle, $message)`
: This callback is called when the connection could not be established. `$!` will contain the relevant error code, and `$message` a message describing it (usually the same as `"$!"`).

  If this callback isn't specified, then `on_error` will be called with a fatal error instead.

on_error => `$cb->($handle, $fatal, $message)`
: This is the error callback, which is called when, well, some error occurred, such as not being able to resolve the hostname, failure to connect, or a read error.

  Some errors are fatal (which is indicated by `$fatal` being true). On fatal errors the handle object will be destroyed (by a call to `-> destroy`) after invoking the error callback (which means you are free to examine the handle object). Examples of fatal errors are an EOF condition with active (but unsatisfiable) read watchers (`EPIPE`) or I/O errors. In cases where the other side can close the connection at will, it is often easiest to not report `EPIPE` errors in this callback.

  AnyEvent::Handle tries to find an appropriate error code for you to check against, but in some cases (TLS errors), this does not work well.

  If you report the error to the user, it is recommended to always output the `$message` argument in human-readable error messages (you don't need to report `"$!"` if you report `$message`).

  If you want to react programmatically to the error, then looking at `$!` and comparing it against some of the documented `Errno` values is usually better than looking at the `$message`.

  Non-fatal errors can be retried by returning, but it is recommended to simply ignore this parameter and instead abondon the handle object when this callback is invoked. Examples of non-fatal errors are timeouts `ETIMEDOUT`) or badly-formatted data (`EBADMSG`).

  On entry to the callback, the value of `$!` contains the operating system error code (or `ENOSPC`, `EPIPE`, `ETIMEDOUT`, `EBADMSG` or `EPROTO`).

  While not mandatory, it is *highly* recommended to set this callback, as you will not be notified of errors otherwise. The default just calls `croak`.

on_read => $cb->($handle)
: This sets the default read callback, which is called when data arrives and no read request is in the queue (unlike read queue callbacks, this callback will only be called when at least one octet of data is in the read buffer).

  To access (and remove data from) the read buffer, use the `->rbuf` method or access the `$handle->{rbuf}` member directly. Note that you must not enlarge or modify the read buffer, you can only remove data at the beginning from it.

  You can also call `->push_read (...)` or any other function that modifies the read queue. Or do both. Or ...

  When an EOF condition is detected, AnyEvent::Handle will first try to feed all the remaining data to the queued callbacks and `on_read` before calling the `on_eof` callback. If no progress can be made, then a fatal error will be raised (with `$!` set to `EPIPE`).

  Note that, unlike requests in the read queue, an `on_read` callback doesn't mean you *require* some data: if there is an EOF and there are outstanding read requests then an error will be flagged. With an `on_read` callback, the `on_eof` callback will be invoked.

on_eof => $cb->($handle)
: Set the callback to be called when an end-of-file condition is detected, i.e. in the case of a socket, when the other side has closed the connection cleanly, and there are no outstanding read requests in the queue (if there are read requests, then an EOF counts as an unexpected connection close and will be flagged as an error).

  For sockets, this just means that the other side has stopped sending data, you can still try to write data, and, in fact, one can return from the EOF callback and continue writing data, as only the read part has been shut down.

  If an EOF condition has been detected but no `on_eof` callback has been set, then a fatal error will be raised with `$!` set to <0>.

on_drain => $cb->($handle)
: This sets the callback that is called once when the write buffer becomes empty (and immediately when the handle object is created).

  To append to the write buffer, use the `->push_write` method.

  This callback is useful when you don't want to put all of your write data into the queue at once, for example, when you want to write the contents of some file to the socket you might not want to read the whole file into memory and push it into the queue, but instead only read more data from the file when the write queue becomes empty.

timeout => $fractional_seconds
rtimeout => $fractional_seconds
wtimeout => $fractional_seconds
: If non-zero, then these enables an ''inactivity'' timeout: whenever this many seconds pass without a successful read or write on the underlying file handle (or a call to `timeout_reset`), the `on_timeout` callback will be invoked (and if that one is missing, a non-fatal `ETIMEDOUT` error will be raised).

  There are three variants of the timeouts that work independently of each other, for both read and write (triggered when nothing was read *OR* written), just read (triggered when nothing was read), and just write: `timeout`, `rtimeout` and `wtimeout`, with corresponding callbacks `on_timeout`, `on_rtimeout` and `on_wtimeout`, and reset functions `timeout_reset`, `rtimeout_reset`, and `wtimeout_reset`.

  Note that timeout processing is active even when you do not have any outstanding read or write requests: If you plan to keep the connection idle then you should disable the timeout temporarily or ignore the timeout in the corresponding `on_timeout` callback, in which case

AnyEvent::Handle will simply restart the timeout.

Zero (the default) disables the corresponding timeout.

on_timeout => `$cb->`($handle)
on_rtimeout => `$cb->`($handle)
on_wtimeout => `$cb->`($handle)
Called whenever the inactivity timeout passes. If you return from this callback, then the timeout will be reset as if some activity had happened, so this condition is not fatal in any way.

rbuf_max => <bytes>
If defined, then a fatal error will be raised (with `$!` set to `ENOSPC`) when the read buffer ever (strictly) exceeds this size. This is useful to avoid some forms of denial-of-service attacks.

For example, a server accepting connections from untrusted sources should be configured to accept only so-and-so much data that it cannot act on (for example, when expecting a line, an attacker could send an unlimited amount of data without a callback ever being called as long as the line isn't finished).

wbuf_max => <bytes>
If defined, then a fatal error will be raised (with `$!` set to `ENOSPC`) when the write buffer ever (strictly) exceeds this size. This is useful to avoid some forms of denial-of-service attacks.

Although the units of this parameter is bytes, this is the *raw* number of bytes not yet accepted by the kernel. This can make a difference when you e.g. use TLS, as TLS typically makes your write data larger (but it can also make it smaller due to compression).

As an example of when this limit is useful, take a chat server that sends chat messages to a client. If the client does not read those in a timely manner then the send buffer in the server would grow unbounded.

autocork => <boolean>
When disabled (the default), `push_write` will try to immediately write the data to the handle if possible. This avoids having to register a write watcher and wait for the next event loop iteration, but can be inefficient if you write multiple small chunks (on the wire, this disadvantage is usually avoided by your kernel's nagle algorithm, see `no_delay`, but this option can save costly syscalls).

When enabled, writes will always be queued till the next event loop iteration. This is efficient when you do many small writes per iteration, but less efficient when you do a single write only per iteration (or when the write buffer often is full). It also increases write latency.

no_delay => <boolean>
When doing small writes on sockets, your operating system kernel might wait a bit for more data before actually sending it out. This is called the Nagle algorithm, and usually it is beneficial.

In some situations you want as low a delay as possible, which can be accomplishd by setting this option to a true value.

The default is your operating system's default behaviour (most likely enabled). This option explicitly enables or disables it, if possible.

keepalive => <boolean>
Enables (default disable) the SO_KEEPALIVE option on the stream socket: normally, TCP connections have no time-out once established, so TCP connections, once established, can stay alive forever even when the other side has long gone. TCP keepalives are a cheap way to take down long-lived TCP connections when the other side becomes unreachable. While the default is OS-dependent, TCP keepalives usually kick in after around two hours, and, if the other side doesn't reply, take down the TCP connection some 10 to 15 minutes later.

It is harmless to specify this option for file handles that do not support keepalives, and enabling it on connections that are potentially long-lived is usually a good idea.

oobinline => <boolean>
: BSD majorly fucked up the implementation of TCP urgent data. The result is that almost no OS implements TCP according to the specs, and every OS implements it slightly differently.

  If you want to handle TCP urgent data, then setting this flag (the default is enabled) gives you the most portable way of getting urgent data, by putting it into the stream.

  Since BSD emulation of OOB data on top of TCP's urgent data can have security implications, AnyEvent::Handle sets this flag automatically unless explicitly specified. Note that setting this flag after establishing a connection *may* be a bit too late (data loss could already have occurred on BSD systems), but at least it will protect you from most attacks.

read_size => <bytes>
: The initial read block size, the number of bytes this module will try to read during each loop iteration. Each handle object will consume at least this amount of memory for the read buffer as well, so when handling many connections watch out for memory requirements). See also `max_read_size`. Default: `2048`.

max_read_size => <bytes>
: The maximum read buffer size used by the dynamic adjustment algorithm: Each time AnyEvent::Handle can read `read_size` bytes in one go it will double `read_size` up to the maximum given by this option. Default: `131072` or `read_size`, whichever is higher.

low_water_mark => <bytes>
: Sets the number of bytes (default: `0`) that make up an ''empty'' write buffer: If the buffer reaches this size or gets even samller it is considered empty.

  Sometimes it can be beneficial (for performance reasons) to add data to the write buffer before it is fully drained, but this is a rare case, as the operating system kernel usually buffers data as well, so the default is good in almost all cases.

linger => <seconds>
: If this is non-zero (default: `3600`), the destructor of the AnyEvent::Handle object will check whether there is still outstanding write data and will install a watcher that will write this data to the socket. No errors will be reported (this mostly matches how the operating system treats outstanding data at socket close time).

  This will not work for partial TLS data that could not be encoded yet. This data will be lost. Calling the `stoptls` method in time might help.

peername => `$string`
: A string used to identify the remote site – usually the DNS hostname (*not* IDN!) used to create the connection, rarely the IP address.

  Apart from being useful in error messages, this string is also used in TLS peername verification (see `verify_peername` in AnyEvent::TLS). This verification will be skipped when `peername` is not specified or is `undef`.

tls => ''accept'' | ''connect'' | Net::SSLeay::SSL object
: When this parameter is given, it enables TLS (SSL) mode, that means AnyEvent will start a TLS handshake as soon as the connection has been established and will transparently encrypt/decrypt data afterwards.

  All TLS protocol errors will be signalled as `EPROTO`, with an appropriate error message.

  TLS mode requires Net::SSLeay to be installed (it will be loaded automatically when you try to create a TLS handle): this module doesn't have a dependency on that module, so if your module requires it, you have to add the dependency yourself. If Net::SSLeay cannot be loaded or is too old, you get an `EPROTO` error.

  Unlike TCP, TLS has a server and client side: for the TLS server side, use `accept`, and for the TLS client side of a connection, use `connect` mode.

You can also provide your own TLS connection object, but you have to make sure that you call either `Net::SSLeay::set_connect_state` or `Net::SSLeay::set_accept_state` on it before you pass it to AnyEvent::Handle. Also, this module will take ownership of this connection object.

At some future point, AnyEvent::Handle might switch to another TLS implementation, then the option to use your own session object will go away.

**IMPORTANT:** since Net::SSLeay "objects" are really only integers, passing in the wrong integer will lead to certain crash. This most often happens when one uses a stylish `tls => 1` and is surprised about the segmentation fault.

Use the `->starttls` method if you need to start TLS negotiation later.

tls_ctx => `$anyevent_tls`

Use the given `AnyEvent::TLS` object to create the new TLS connection (unless a connection object was specified directly). If this parameter is missing (or `undef`), then AnyEvent::Handle will use `AnyEvent::Handle::TLS_CTX`.

Instead of an object, you can also specify a hash reference with `key => value` pairs. Those will be passed to AnyEvent::TLS to create a new TLS context object.

on_starttls => `$cb->`($handle, `$success`[, `$error_message`])

This callback will be invoked when the TLS/SSL handshake has finished. If `$success` is true, then the TLS handshake succeeded, otherwise it failed (`on_stoptls` will not be called in this case).

The session in `$handle->{tls}` can still be examined in this callback, even when the handshake was not successful.

TLS handshake failures will not cause `on_error` to be invoked when this callback is in effect, instead, the error message will be passed to `on_starttls`.

Without this callback, handshake failures lead to `on_error` being called as usual.

Note that you cannot just call `starttls` again in this callback. If you need to do that, start an zero-second timer instead whose callback can then call `->starttls` again.

on_stoptls => `$cb->`($handle)

When a SSLv3/TLS shutdown/close notify/EOF is detected and this callback is set, then it will be invoked after freeing the TLS session. If it is not, then a TLS shutdown condition will be treated like a normal EOF condition on the handle.

The session in `$handle->{tls}` can still be examined in this callback.

This callback will only be called on TLS shutdowns, not when the underlying handle signals EOF.

json => JSON, JSON::PP or JSON::XS object

This is the json coder object used by the `json` read and write types.

If you don't supply it, then AnyEvent::Handle will create and use a suitable one (on demand), which will write and expect UTF-8 encoded JSON texts (either using JSON::XS or JSON). The written texts are guaranteed not to contain any newline character.

For security reasons, this encoder will likely *not* handle numbers and strings, only arrays and objects/hashes. The reason is that originally JSON was self-delimited, but Dougles Crockford thought it was a splendid idea to redefine JSON incompatibly, so this is no longer true.

For protocols that used back-to-back JSON texts, this might lead to run-ins, where two or more JSON texts will be interpreted as one JSON text.

For this reason, if the default encoder uses JSON::XS, it will default to not allowing anything but arrays and objects/hashes, at least for the foreseeable future (it will change at some point). This might or might not be true for the JSON module, so this might cause a security issue.

If you depend on either behaviour, you should create your own json object and pass it in explicitly.

cbor => CBOR::XS object
: This is the cbor coder object used by the `cbor` read and write types.

  If you don't supply it, then AnyEvent::Handle will create and use a suitable one (on demand), which will write CBOR without using extensions, if possible.

  Note that you are responsible to depend on the CBOR::XS module if you want to use this functionality, as AnyEvent does not have a dependency on it itself.

`$fh = $handle->fh`
: This method returns the file handle used to create the AnyEvent::Handle object.

`$handle->on_error ($cb)`
: Replace the current `on_error` callback (see the `on_error` constructor argument).

`$handle->on_eof ($cb)`
: Replace the current `on_eof` callback (see the `on_eof` constructor argument).

`$handle->on_timeout ($cb)`
`$handle->on_rtimeout ($cb)`
`$handle->on_wtimeout ($cb)`
: Replace the current `on_timeout`, `on_rtimeout` or `on_wtimeout` callback, or disables the callback (but not the timeout) if `$cb = undef`. See the `timeout` constructor argument and method.

`$handle->autocork ($boolean)`
: Enables or disables the current autocork behaviour (see `autocork` constructor argument). Changes will only take effect on the next write.

`$handle->no_delay ($boolean)`
: Enables or disables the `no_delay` setting (see constructor argument of the same name for details).

`$handle->keepalive ($boolean)`
: Enables or disables the `keepalive` setting (see constructor argument of the same name for details).

`$handle->oobinline ($boolean)`
: Enables or disables the `oobinline` setting (see constructor argument of the same name for details).

`$handle->on_starttls ($cb)`
: Replace the current `on_starttls` callback (see the `on_starttls` constructor argument).

`$handle->on_stoptls ($cb)`
: Replace the current `on_stoptls` callback (see the `on_stoptls` constructor argument).

`$handle->rbuf_max ($max_octets)`
: Configures the `rbuf_max` setting (`undef` disables it).

`$handle->wbuf_max ($max_octets)`
: Configures the `wbuf_max` setting (`undef` disables it).

`$handle->timeout ($seconds)`
`$handle->rtimeout ($seconds)`
`$handle->wtimeout ($seconds)`
: Configures (or disables) the inactivity timeout.

  The timeout will be checked instantly, so this method might destroy the handle before it returns.

`$handle->timeout_reset`
`$handle->rtimeout_reset`
`$handle->wtimeout_reset`
: Reset the activity timeout, as if data was received or sent.

  These methods are cheap to call.

### WRITE QUEUE

AnyEvent::Handle manages two queues per handle, one for writing and one for reading.

The write queue is very simple: you can add data to its end, and AnyEvent::Handle will automatically try to get rid of it for you.

When data could be written and the write buffer is shorter then the low water mark, the `on_drain` callback will be invoked once.

`$handle->on_drain ($cb)`

Sets the `on_drain` callback or clears it (see the description of `on_drain` in the constructor).

This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

`$handle->push_write ($data)`

Queues the given scalar to be written. You can push as much data as you want (only limited by the available memory and `wbuf_max`), as `AnyEvent::Handle` buffers it independently of the kernel.

This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

`$handle->push_write (type => @args)`

Instead of formatting your data yourself, you can also let this module do the job by specifying a type and type-specific arguments. You can also specify the (fully qualified) name of a package, in which case AnyEvent tries to load the package and then expects to find the `anyevent_write_type` function inside (see "custom write types", below).

Predefined types are (if you have ideas for additional types, feel free to drop by and tell us):

netstring => `$string`

Formats the given value as netstring (http://cr.yp.to/proto/netstrings.txt, this is not a recommendation to use them).

packstring => `$format`, `$data`

An octet string prefixed with an encoded length. The encoding `$format` uses the same format as a Perl `pack` format, but must specify a single integer only (only one of `cCsSlLqQiInNvVjJw` is allowed, plus an optional `!`, < or > modifier).

json => `$array_or_hashref`

Encodes the given hash or array reference into a JSON object. Unless you provide your own JSON object, this means it will be encoded to JSON text in UTF-8.

The default encoder might or might not handle every type of JSON value - it might be limited to arrays and objects for security reasons. See the `json` constructor attribute for more details.

JSON objects (and arrays) are self-delimiting, so if you only use arrays and hashes, you can write JSON at one end of a handle and read them at the other end without using any additional framing.

The JSON text generated by the default encoder is guaranteed not to contain any newlines: While this module doesn't need delimiters after or between JSON texts to be able to read them, many other languages depend on them.

A simple RPC protocol that interoperates easily with other languages is to send JSON arrays (or objects, although arrays are usually the better choice as they mimic how function argument passing works) and a newline after each JSON text:

```
$handle->push_write (json => ["method", "arg1", "arg2"]); # whatever
$handle->push_write ("\012");
```

An AnyEvent::Handle receiver would simply use the `json` read type and rely on the fact that the newline will be skipped as leading whitespace:

```
$handle->push_read (json => sub { my $array = $_[1]; ... });
```

Other languages could read single lines terminated by a newline and pass this line into their JSON decoder of choice.

cbor => `$perl_scalar`
: Encodes the given scalar into a CBOR value. Unless you provide your own CBOR::XS object, this means it will be encoded to a CBOR string not using any extensions, if possible.

  CBOR values are self-delimiting, so you can write CBOR at one end of a handle and read them at the other end without using any additional framing.

  A simple nd very very fast RPC protocol that interoperates with other languages is to send CBOR and receive CBOR values (arrays are recommended):

```
$handle->push_write (cbor => ["method", "arg1", "arg2"]); # whatever
```

  An AnyEvent::Handle receiver would simply use the `cbor` read type:

```
$handle->push_read (cbor => sub { my $array = $_[1]; ... });
```

storable => `$reference`
: Freezes the given reference using Storable and writes it to the handle. Uses the `nfreeze` format.

`$handle->`push_shutdown
: Sometimes you know you want to close the socket after writing your data before it was actually written. One way to do that is to replace your `on_drain` handler by a callback that shuts down the socket (and set `low_water_mark` to `0`). This method is a shorthand for just that, and replaces the `on_drain` callback with:

```
sub { shutdown $_[0]{fh}, 1 }
```

  This simply shuts down the write side and signals an EOF condition to the the peer.

  You can rely on the normal read queue and `on_eof` handling afterwards. This is the cleanest way to close a connection.

  This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

custom write types – Package::anyevent_write_type `$handle`, `@args`
: Instead of one of the predefined types, you can also specify the name of a package. AnyEvent will try to load the package and then expects to find a function named `anyevent_write_type` inside. If it isn't found, it progressively tries to load the parent package until it either finds the function (good) or runs out of packages (bad).

  Whenever the given `type` is used, `push_write` will the function with the handle object and the remaining arguments.

  The function is supposed to return a single octet string that will be appended to the write buffer, so you can mentally treat this function as a "arguments to on-the-wire-format" converter.

  Example: implement a custom write type `join` that joins the remaining arguments using the first one.

```
$handle->push_write (My::Type => " ", 1,2,3);

# uses the following package, which can be defined in the "My::Type" or in
# the "My" modules to be auto-loaded, or just about anywhere when the
# My::Type::anyevent_write_type is defined before invoking it.

package My::Type;

sub anyevent_write_type {
   my ($handle, $delim, @args) = @_;

   join $delim, @args
}
```

### READ QUEUE

AnyEvent::Handle manages two queues per handle, one for writing and one for reading.

The read queue is more complex than the write queue. It can be used in two ways, the "simple" way, using only `on_read` and the "complex" way, using a queue.

In the simple case, you just install an `on_read` callback and whenever new data arrives, it will be called. You can then remove some data (if enough is there) from the read buffer (`$handle->rbuf`). Or you can leave the data there if you want to accumulate more (e.g. when only a partial message has been received so far), or change the read queue with e.g. `push_read`.

In the more complex case, you want to queue multiple callbacks. In this case, AnyEvent::Handle will call the first queued callback each time new data arrives (also the first time it is queued) and remove it when it has done its job (see `push_read`, below).

This way you can, for example, push three line-reads, followed by reading a chunk of data, and AnyEvent::Handle will execute them in order.

Example 1: EPP protocol parser. EPP sends 4 byte length info, followed by the specified number of bytes which give an XML datagram.

```
# in the default state, expect some header bytes
$handle->on_read (sub {
   # some data is here, now queue the length-header-read (4 octets)
   shift->unshift_read (chunk => 4, sub {
      # header arrived, decode
      my $len = unpack "N", $_[1];

      # now read the payload
      shift->unshift_read (chunk => $len, sub {
         my $xml = $_[1];
         # handle xml
      });
   });
});
```

Example 2: Implement a client for a protocol that replies either with "OK" and another line or "ERROR" for the first request that is sent, and 64 bytes for the second request. Due to the availability of a queue, we can just pipeline sending both requests and manipulate the queue as necessary in the callbacks.

When the first callback is called and sees an "OK" response, it will `unshift` another line-read. This line-read will be queued *before* the 64-byte chunk callback.

```
# request one, returns either "OK + extra line" or "ERROR"
$handle->push_write ("request 1\015\012");

# we expect "ERROR" or "OK" as response, so push a line read
$handle->push_read (line => sub {
   # if we got an "OK", we have to _prepend_ another line,
   # so it will be read before the second request reads its 64 bytes
   # which are already in the queue when this callback is called
   # we don't do this in case we got an error
   if ($_[1] eq "OK") {
      $_[0]->unshift_read (line => sub {
         my $response = $_[1];
         ...
      });
   }
});
```

```
   # request two, simply returns 64 octets
   $handle->push_write ("request 2\015\012");

   # simply read 64 bytes, always
   $handle->push_read (chunk => 64, sub {
      my $response = $_[1];
      ...
   });
```

$handle->on_read ($cb)
: This replaces the currently set `on_read` callback, or clears it (when the new callback is `undef`). See the description of `on_read` in the constructor.

  This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

$handle->rbuf
: Returns the read buffer (as a modifiable lvalue). You can also access the read buffer directly as the `->{rbuf}` member, if you want (this is much faster, and no less clean).

  The only operation allowed on the read buffer (apart from looking at it) is removing data from its beginning. Otherwise modifying or appending to it is not allowed and will lead to hard-to-track-down bugs.

  NOTE: The read buffer should only be used or modified in the `on_read` callback or when `push_read` or `unshift_read` are used with a single callback (i.e. untyped). Typed `push_read` and `unshift_read` methods will manage the read buffer on their own.

$handle->push_read ($cb)
$handle->unshift_read ($cb)
: Append the given callback to the end of the queue (`push_read`) or prepend it (`unshift_read`).

  The callback is called each time some additional read data arrives.

  It must check whether enough data is in the read buffer already.

  If not enough data is available, it must return the empty list or a false value, in which case it will be called repeatedly until enough data is available (or an error condition is detected).

  If enough data was available, then the callback must remove all data it is interested in (which can be none at all) and return a true value. After returning true, it will be removed from the queue.

  These methods may invoke callbacks (and therefore the handle might be destroyed after it returns).

$handle->push_read (type => `@args`, `$cb`)
$handle->unshift_read (type => `@args`, `$cb`)
: Instead of providing a callback that parses the data itself you can chose between a number of predefined parsing formats, for chunks of data, lines etc. You can also specify the (fully qualified) name of a package, in which case AnyEvent tries to load the package and then expects to find the `anyevent_read_type` function inside (see "custom read types", below).

  Predefined types are (if you have ideas for additional types, feel free to drop by and tell us):

  chunk => `$octets`, `$cb`->($handle, `$data`)
  : Invoke the callback only once `$octets` bytes have been read. Pass the data read to the callback. The callback will never be called with less data.

    Example: read 2 bytes.

    ```
    $handle->push_read (chunk => 2, sub {
       say "yay " . unpack "H*", $_[1];
    });
    ```

line => [$eol, ]$cb->($handle, `$line`, `$eol`)
The callback will be called only once a full line (including the end of line marker, `$eol`) has been read. This line (excluding the end of line marker) will be passed to the callback as second argument (`$line`), and the end of line marker as the third argument (`$eol`).

The end of line marker, `$eol`, can be either a string, in which case it will be interpreted as a fixed record end marker, or it can be a regex object (e.g. created by `qr`), in which case it is interpreted as a regular expression.

The end of line marker argument `$eol` is optional, if it is missing (NOT undef), then `qr|\015?\012|` is used (which is good for most internet protocols).

Partial lines at the end of the stream will never be returned, as they are not marked by the end of line marker.

regex => `$accept`[, `$reject`[, `$skip`], `$cb`->($handle, `$data`)
Makes a regex match against the regex object `$accept` and returns everything up to and including the match. All the usual regex variables ($1, %+ etc.) from the regex match are available in the callback.

Example: read a single line terminated by '\n'.

```
$handle->push_read (regex => qr<\n>, sub { ... });
```

If `$reject` is given and not undef, then it determines when the data is to be rejected: it is matched against the data when the `$accept` regex does not match and generates an `EBADMSG` error when it matches. This is useful to quickly reject wrong data (to avoid waiting for a timeout or a receive buffer overflow).

Example: expect a single decimal number followed by whitespace, reject anything else (not the use of an anchor).

```
$handle->push_read (regex => qr<^[0-9]+\s>, qr<[^0-9]>, sub { ... });
```

If `$skip` is given and not `undef`, then it will be matched against the receive buffer when neither `$accept` nor `$reject` match, and everything preceding and including the match will be accepted unconditionally. This is useful to skip large amounts of data that you know cannot be matched, so that the `$accept` or `$reject` regex do not have to start matching from the beginning. This is purely an optimisation and is usually worth it only when you expect more than a few kilobytes.

Example: expect a http header, which ends at `\015\012\015\012`. Since we expect the header to be very large (it isn't in practice, but...), we use a skip regex to skip initial portions. The skip regex is tricky in that it only accepts something not ending in either \015 or \012, as these are required for the accept regex.

```
$handle->push_read (regex =>
   qr<\015\012\015\012>,
   undef, # no reject
   qr<^.*[^\015\012]>,
   sub { ... });
```

netstring => `$cb`->($handle, `$string`)
A netstring (http://cr.yp.to/proto/netstrings.txt, this is not an endorsement).

Throws an error with `$!` set to EBADMSG on format violations.

packstring => `$format`, `$cb`->($handle, `$string`)
An octet string prefixed with an encoded length. The encoding `$format` uses the same format as a Perl `pack` format, but must specify a single integer only (only one of `cCsSlLqQiInNvVjJw` is allowed, plus an optional `!`, < or > modifier).

For example, DNS over TCP uses a prefix of `n` (2 octet network order), EPP uses a prefix of `N` (4

octtes).

Example: read a block of data prefixed by its length in BER-encoded format (very efficient).

```
$handle->push_read (packstring => "w", sub {
   my ($handle, $data) = @_;
});
```

json => $cb->($handle, $hash_or_arrayref)
Reads a JSON object or array, decodes it and passes it to the callback. When a parse error occurs, an EBADMSG error will be raised.

If a json object was passed to the constructor, then that will be used for the final decode, otherwise it will create a JSON::XS or JSON::PP coder object expecting UTF-8.

This read type uses the incremental parser available with JSON version 2.09 (and JSON::XS version 2.2) and above.

Since JSON texts are fully self-delimiting, the json read and write types are an ideal simple RPC protocol: just exchange JSON datagrams. See the json write type description, above, for an actual example.

cbor => $cb->($handle, $scalar)
Reads a CBOR value, decodes it and passes it to the callback. When a parse error occurs, an EBADMSG error will be raised.

If a CBOR::XS object was passed to the constructor, then that will be used for the final decode, otherwise it will create a CBOR coder without enabling any options.

You have to provide a dependency to CBOR::XS on your own: this module will load the CBOR::XS module, but AnyEvent does not depend on it itself.

Since CBOR values are fully self-delimiting, the cbor read and write types are an ideal simple RPC protocol: just exchange CBOR datagrams. See the cbor write type description, above, for an actual example.

storable => $cb->($handle, $ref)
Deserialises a Storable frozen representation as written by the storable write type (BER-encoded length prefix followed by nfreeze'd data).

Raises EBADMSG error if the data could not be decoded.

tls_detect => $cb->($handle, $detect, $major, $minor)
Checks the input stream for a valid SSL or TLS handshake TLSPaintext record without consuming anything. Only SSL version 3 or higher is handled, up to the fictituous protocol 4.x (but both SSL3+ and SSL2-compatible framing is supported).

If it detects that the input data is likely TLS, it calls the callback with a true value for $detect and the (on-wire) TLS version as second and third argument ($major is 3, and $minor is 0..4 for SSL 3.0, TLS 1.0, 1.1, 1.2 and 1.3, respectively). If it detects the input to be definitely not TLS, it calls the callback with a false value for $detect.

The callback could use this information to decide whether or not to start TLS negotiation.

In all cases the data read so far is passed to the following read handlers.

Usually you want to use the tls_autostart read type instead.

If you want to design a protocol that works in the presence of TLS dtection, make sure that any non-TLS data doesn't start with the octet 22 (ASCII SYN, 16 hex) or 128-255 (i.e. highest bit set). The checks this read type does are a bit more strict, but might losen in the future to accommodate protocol changes.

This read type does not rely on AnyEvent::TLS (and thus, not on Net::SSLeay).

tls_autostart => [$tls_ctx, ]$tls
Tries to detect a valid SSL or TLS handshake. If one is detected, it tries to start tls by calling `starttls` with the given arguments.

In practise, `$tls` must be `accept`, or a Net::SSLeay context that has been configured to accept, as servers do not normally send a handshake on their own and ths cannot be detected in this way.

See `tls_detect` above for more details.

Example: give the client a chance to start TLS before accepting a text line.

```
$hdl->push_read (tls_autostart => "accept");
$hdl->push_read (line => sub {
   print "received ", ($_[0]{tls} ? "encrypted" : "cleartext"), " <$_[1]
});
```

custom read types – Package::anyevent_read_type `$handle`, `$cb`, `@args`
Instead of one of the predefined types, you can also specify the name of a package. AnyEvent will try to load the package and then expects to find a function named `anyevent_read_type` inside. If it isn't found, it progressively tries to load the parent package until it either finds the function (good) or runs out of packages (bad).

Whenever this type is used, `push_read` will invoke the function with the handle object, the original callback and the remaining arguments.

The function is supposed to return a callback (usually a closure) that works as a plain read callback (see `->push_read ($cb)`), so you can mentally treat the function as a "configurable read type to read callback" converter.

It should invoke the original callback when it is done reading (remember to pass `$handle` as first argument as all other callbacks do that, although there is no strict requirement on this).

For examples, see the source of this module (*perldoc -m AnyEvent::Handle*, search for `register_read_type`)).

`$handle->`stop_read
`$handle->`start_read
In rare cases you actually do not want to read anything from the socket. In this case you can call `stop_read`. Neither `on_read` nor any queued callbacks will be executed then. To start reading again, call `start_read`.

Note that AnyEvent::Handle will automatically `start_read` for you when you change the `on_read` callback or push/unshift a read callback, and it will automatically `stop_read` for you when neither `on_read` is set nor there are any read requests in the queue.

In older versions of this module (<= 5.3), these methods had no effect, as TLS does not support half-duplex connections. In current versions they work as expected, as this behaviour is required to avoid certain resource attacks, where the program would be forced to read (and buffer) arbitrary amounts of data before being able to send some data. The drawback is that some readings of the the SSL/TLS specifications basically require this attack to be working, as SSL/TLS implementations might stall sending data during a rehandshake.

As a guideline, during the initial handshake, you should not stop reading, and as a client, it might cause problems, depending on your application.

`$handle->`starttls ($tls[, `$tls_ctx`])
Instead of starting TLS negotiation immediately when the AnyEvent::Handle object is created, you can also do that at a later time by calling `starttls`. See the `tls` constructor argument for general info.

Starting TLS is currently an asynchronous operation – when you push some write data and then call `->starttls` then TLS negotiation will start immediately, after which the queued write data is then sent. This might change in future versions, so best make sure you have no outstanding write data when

calling this method.

The first argument is the same as the `tls` constructor argument (either `"connect"`, `"accept"` or an existing Net::SSLeay object).

The second argument is the optional `AnyEvent::TLS` object that is used when AnyEvent::Handle has to create its own TLS connection object, or a hash reference with `key => value` pairs that will be used to construct a new context.

The TLS connection object will end up in `$handle->{tls}`, the TLS context in `$handle->{tls_ctx}` after this call and can be used or changed to your liking. Note that the handshake might have already started when this function returns.

Due to bugs in OpenSSL, it might or might not be possible to do multiple handshakes on the same stream. It is best to not attempt to use the stream after stopping TLS.

This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

`$handle->stoptls`
Shuts down the SSL connection - this makes a proper EOF handshake by sending a close notify to the other side, but since OpenSSL doesn't support non-blocking shut downs, it is not guaranteed that you can re-use the stream afterwards.

This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

`$handle->resettls`
This rarely-used method simply resets and TLS state on the handle, usually causing data loss.

One case where it may be useful is when you want to skip over the data in the stream but you are not interested in interpreting it, so data loss is no concern.

`$handle->destroy`
Shuts down the handle object as much as possible - this call ensures that no further callbacks will be invoked and as many resources as possible will be freed. Any method you will call on the handle object after destroying it in this way will be silently ignored (and it will return the empty list).

Normally, you can just "forget" any references to an AnyEvent::Handle object and it will simply shut down. This works in fatal error and EOF callbacks, as well as code outside. It does *NOT* work in a read or write callback, so when you want to destroy the AnyEvent::Handle object from within such an callback. You *MUST* call `->destroy` explicitly in that case.

Destroying the handle object in this way has the advantage that callbacks will be removed as well, so if those are the only reference holders (as is common), then one doesn't need to do anything special to break any reference cycles.

The handle might still linger in the background and write out remaining data, as specified by the `linger` option, however.

`$handle->destroyed`
Returns false as long as the handle hasn't been destroyed by a call to `->destroy`, true otherwise.

Can be useful to decide whether the handle is still valid after some callback possibly destroyed the handle. For example, `->push_write`, `->starttls` and other methods can call user callbacks, which in turn can destroy the handle, so work can be avoided by checking sometimes:

```
$hdl->starttls ("accept");
return if $hdl->destroyed;
$hdl->push_write (...
```

Note that the call to `push_write` will silently be ignored if the handle has been destroyed, so often you can just ignore the possibility of the handle being destroyed.

AnyEvent::Handle::TLS_CTX
This function creates and returns the AnyEvent::TLS object used by default for TLS mode.

The context is created by calling AnyEvent::TLS without any arguments.

## NONFREQUENTLY ASKED QUESTIONS

I `undef` the AnyEvent::Handle reference inside my callback and still get further invocations!
That's because AnyEvent::Handle keeps a reference to itself when handling read or write callbacks.

It is only safe to "forget" the reference inside EOF or error callbacks, from within all other callbacks, you need to explicitly call the `->destroy` method.

Why is my `on_eof` callback never called?
Probably because your `on_error` callback is being called instead: When you have outstanding requests in your read queue, then an EOF is considered an error as you clearly expected some data.

To avoid this, make sure you have an empty read queue whenever your handle is supposed to be "idle" (i.e. connection closes are O.K.). You can set an `on_read` handler that simply pushes the first read requests in the queue.

See also the next question, which explains this in a bit more detail.

How can I serve requests in a loop?
Most protocols consist of some setup phase (authentication for example) followed by a request handling phase, where the server waits for requests and handles them, in a loop.

There are two important variants: The first (traditional, better) variant handles requests until the server gets some QUIT command, causing it to close the connection first (highly desirable for a busy TCP server). A client dropping the connection is an error, which means this variant can detect an unexpected detection close.

To handle this case, always make sure you have a non-empty read queue, by pushing the "read request start" handler on it:

```
# we assume a request starts with a single line
my @start_request; @start_request = (line => sub {
   my ($hdl, $line) = @_;

   ... handle request

   # push next request read, possibly from a nested callback
   $hdl->push_read (@start_request);
});

# auth done, now go into request handling loop
# now push the first @start_request
$hdl->push_read (@start_request);
```

By always having an outstanding `push_read`, the handle always expects some data and raises the `EPIPE` error when the connction is dropped unexpectedly.

The second variant is a protocol where the client can drop the connection at any time. For TCP, this means that the server machine may run out of sockets easier, and in general, it means you cannot distinguish a protocl failure/client crash from a normal connection close. Nevertheless, these kinds of protocols are common (and sometimes even the best solution to the problem).

Having an outstanding read request at all times is possible if you ignore `EPIPE` errors, but this doesn't help with when the client drops the connection during a request, which would still be an error.

A better solution is to push the initial request read in an `on_read` callback. This avoids an error, as when the server doesn't expect data (i.e. is idly waiting for the next request, an EOF will not raise an error, but simply result in an `on_eof` callback. It is also a bit slower and simpler:

```
   # auth done, now go into request handling loop
   $hdl->on_read (sub {
      my ($hdl) = @_;

      # called each time we receive data but the read queue is empty
      # simply start read the request

      $hdl->push_read (line => sub {
         my ($hdl, $line) = @_;

         ... handle request

         # do nothing special when the request has been handled, just
         # let the request queue go empty.
      });
   });
```

I get different callback invocations in TLS mode/Why can't I pause reading?

Unlike, say, TCP, TLS connections do not consist of two independent communication channels, one for each direction. Or put differently, the read and write directions are not independent of each other: you cannot write data unless you are also prepared to read, and vice versa.

This means that, in TLS mode, you might get `on_error` or `on_eof` callback invocations when you are not expecting any read data – the reason is that AnyEvent::Handle always reads in TLS mode.

During the connection, you have to make sure that you always have a non-empty read-queue, or an `on_read` watcher. At the end of the connection (or when you no longer want to use it) you can call the `destroy` method.

How do I read data until the other side closes the connection?

If you just want to read your data into a perl scalar, the easiest way to achieve this is by setting an `on_read` callback that does nothing, clearing the `on_eof` callback and in the `on_error` callback, the data will be in `$_[0]{rbuf}`:

```
   $handle->on_read (sub { });
   $handle->on_eof (undef);
   $handle->on_error (sub {
      my $data = delete $_[0]{rbuf};
   });
```

Note that this example removes the `rbuf` member from the handle object, which is not normally allowed by the API. It is expressly permitted in this case only, as the handle object needs to be destroyed afterwards.

The reason to use `on_error` is that TCP connections, due to latencies and packets loss, might get closed quite violently with an error, when in fact all data has been received.

It is usually better to use acknowledgements when transferring data, to make sure the other side hasn't just died and you got the data intact. This is also one reason why so many internet protocols have an explicit QUIT command.

I don't want to destroy the handle too early – how do I wait until all data has been written?

After writing your last bits of data, set the `on_drain` callback and destroy the handle in there – with the default setting of `low_water_mark` this will be called precisely when all data has been written to the socket:

```
$handle->push_write (...);
$handle->on_drain (sub {
   AE::log debug => "All data submitted to the kernel.";
   undef $handle;
});
```

If you just want to queue some data and then signal EOF to the other side, consider using `->push_shutdown` instead.

I want to contact a TLS/SSL server, I don't care about security.

If your TLS server is a pure TLS server (e.g. HTTPS) that only speaks TLS, connect to it and then create the AnyEvent::Handle with the `tls` parameter:

```
tcp_connect $host, $port, sub {
   my ($fh) = @_;

   my $handle = new AnyEvent::Handle
      fh  => $fh,
      tls => "connect",
      on_error => sub { ... };

   $handle->push_write (...);
};
```

I want to contact a TLS/SSL server, I do care about security.

Then you should additionally enable certificate verification, including peername verification, if the protocol you use supports it (see AnyEvent::TLS, `verify_peername`).

E.g. for HTTPS:

```
tcp_connect $host, $port, sub {
   my ($fh) = @_;

    my $handle = new AnyEvent::Handle
       fh       => $fh,
       peername => $host,
       tls      => "connect",
       tls_ctx  => { verify => 1, verify_peername => "https" },
       ...
```

Note that you must specify the hostname you connected to (or whatever "peername" the protocol needs) as the `peername` argument, otherwise no peername verification will be done.

The above will use the system-dependent default set of trusted CA certificates. If you want to check against a specific CA, add the `ca_file` (or `ca_cert`) arguments to `tls_ctx`:

```
tls_ctx  => {
   verify          => 1,
   verify_peername => "https",
   ca_file         => "my-ca-cert.pem",
},
```

I want to create a TLS/SSL server, how do I do that?

Well, you first need to get a server certificate and key. You have three options: a) ask a CA (buy one, use cacert.org etc.) b) create a self-signed certificate (cheap. check the search engine of your choice, there are many tutorials on the net) or c) make your own CA (tinyca2 is a nice program for that purpose).

Then create a file with your private key (in PEM format, see AnyEvent::TLS), followed by the certificate (also in PEM format). The file should then look like this:

```
-----BEGIN RSA PRIVATE KEY-----
...header data
... lots of base64'y-stuff
-----END RSA PRIVATE KEY-----

-----BEGIN CERTIFICATE-----
... lots of base64'y-stuff
-----END CERTIFICATE-----
```

The important bits are the "PRIVATE KEY" and "CERTIFICATE" parts. Then specify this file as `cert_file`:

```
tcp_server undef, $port, sub {
   my ($fh) = @_;

   my $handle = new AnyEvent::Handle
      fh       => $fh,
      tls      => "accept",
      tls_ctx  => { cert_file => "my-server-keycert.pem" },
      ...
```

When you have intermediate CA certificates that your clients might not know about, just append them to the `cert_file`.

## SUBCLASSING AnyEvent::Handle

In many cases, you might want to subclass AnyEvent::Handle.

To make this easier, a given version of AnyEvent::Handle uses these conventions:

- all constructor arguments become object members.

  At least initially, when you pass a `tls`-argument to the constructor it will end up in `$handle->{tls}`. Those members might be changed or mutated later on (for example `tls` will hold the TLS connection object).

- other object member names are prefixed with an _.

  All object members not explicitly documented (internal use) are prefixed with an underscore character, so the remaining non-_-namespace is free for use for subclasses.

- all members not documented here and not prefixed with an underscore are free to use in subclasses.

  Of course, new versions of AnyEvent::Handle may introduce more "public" member variables, but that's just life. At least it is documented.

## AUTHOR

Robin Redeker <elmex at ta-sa.org>, Marc Lehmann <schmorp@schmorp.de>.