NAME

AnyEvent::Log – simple logging "framework"

SYNOPSIS

Simple uses:

```
use AnyEvent;

AE::log fatal => "No config found, cannot continue!"; # never returns
AE::log alert => "The battery died!";
AE::log crit  => "The battery temperature is too hot!";
AE::log error => "Division by zero attempted.";
AE::log warn  => "Couldn't delete the file.";
AE::log note  => "Wanted to create config, but config already exists.";
AE::log info  => "File soandso successfully deleted.";
AE::log debug => "the function returned 3";
AE::log trace => "going to call function abc";
```

Log level overview:

```
LVL NAME      SYSLOG   PERL  NOTE
 1  fatal     emerg    exit  system unusable, aborts program!
 2  alert                    failure in primary system
 3  critical  crit           failure in backup system
 4  error     err      die   non-urgent program errors, a bug
 5  warn      warning        possible problem, not necessarily error
 6  note      notice         unusual conditions
 7  info                     normal messages, no action required
 8  debug                    debugging messages for development
 9  trace                    copious tracing output
```

"Complex" uses (for speed sensitive code, e.g. trace/debug messages):

```
use AnyEvent::Log;

my $tracer = AnyEvent::Log::logger trace => \my $trace;

$tracer->("i am here") if $trace;
$tracer->(sub { "lots of data: " . Dumper $self }) if $trace;
```

Configuration (also look at the EXAMPLES section):

```
# set default logging level to suppress anything below "notice"
# i.e. enable logging at "notice" or above - the default is to
# to not log anything at all.
$AnyEvent::Log::FILTER->level ("notice");

# set logging for the current package to errors and higher only
AnyEvent::Log::ctx->level ("error");

# enable logging for the current package, regardless of global logging level
AnyEvent::Log::ctx->attach ($AnyEvent::Log::LOG);

# enable debug logging for module some::mod and enable logging by default
(AnyEvent::Log::ctx "some::mod")->level ("debug");
(AnyEvent::Log::ctx "some::mod")->attach ($AnyEvent::Log::LOG);

# send all critical and higher priority messages to syslog,
# regardless of (most) other settings
```

```
   $AnyEvent::Log::COLLECT->attach (new AnyEvent::Log::Ctx
      level         => "critical",
      log_to_syslog => "user",
   );
```

DESCRIPTION

This module implements a relatively simple "logging framework". It doesn't attempt to be "the" logging solution or even "a" logging solution for AnyEvent − AnyEvent simply creates logging messages internally, and this module more or less exposes the mechanism, with some extra spiff to allow using it from other modules as well.

Remember that the default verbosity level is `4` (`error`), so only errors and more important messages will be logged, unless you set `PERL_ANYEVENT_VERBOSE` to a higher number before starting your program (`AE_VERBOSE=5` is recommended during development), or change the logging level at runtime with something like:

```
   use AnyEvent::Log;
   $AnyEvent::Log::FILTER->level ("info");
```

The design goal behind this module was to keep it simple (and small), but make it powerful enough to be potentially useful for any module, and extensive enough for the most common tasks, such as logging to multiple targets, or being able to log into a database.

The module is also usable before AnyEvent itself is initialised, in which case some of the functionality might be reduced.

The amount of documentation might indicate otherwise, but the runtime part of the module is still just below 300 lines of code.

LOGGING LEVELS

Logging levels in this module range from `1` (highest priority) to `9` (lowest priority). Note that the lowest numerical value is the highest priority, so when this document says "higher priority" it means "lower numerical value".

Instead of specifying levels by name you can also specify them by aliases:

```
   LVL NAME      SYSLOG   PERL  NOTE
    1  fatal     emerg    exit  system unusable, aborts program!
    2  alert                    failure in primary system
    3  critical  crit           failure in backup system
    4  error     err      die   non-urgent program errors, a bug
    5  warn      warning        possible problem, not necessarily error
    6  note      notice         unusual conditions
    7  info                     normal messages, no action required
    8  debug                    debugging messages for development
    9  trace                    copious tracing output
```

As you can see, some logging levels have multiple aliases − the first one is the "official" name, the second one the "syslog" name (if it differs) and the third one the "perl" name, suggesting (only!) that you log `die` messages at `error` priority. The NOTE column tries to provide some rationale on how to chose a logging level.

As a rough guideline, levels 1..3 are primarily meant for users of the program (admins, staff), and are the only ones logged to STDERR by default. Levels 4..6 are meant for users and developers alike, while levels 7..9 are usually meant for developers.

You can normally only log a message once at highest priority level (`1`, `fatal`), because logging a fatal message will also quit the program − so use it sparingly :)

For example, a program that finds an unknown switch on the commandline might well use a fatal logging level to tell users about it − the "system" in this case would be the program, or module.

Some methods also offer some extra levels, such as `0`, `off`, `none` or `all` − these are only valid for the methods that documented them.

LOGGING FUNCTIONS

The following functions allow you to log messages. They always use the caller's package as a "logging context". Also, the main logging function, `log`, is aliased to `AnyEvent::log` and `AE::log` when the `AnyEvent` module is loaded.

AnyEvent::Log::log `$level`, `$msg`[, `@args`]

Requests logging of the given `$msg` with the given log level, and returns true if the message was logged *somewhere*.

For loglevel `fatal`, the program will abort.

If only a `$msg` is given, it is logged as-is. With extra `@args`, the `$msg` is interpreted as an sprintf format string.

The `$msg` should not end with `\n`, but may if that is convenient for you. Also, multiline messages are handled properly.

Last not least, `$msg` might be a code reference, in which case it is supposed to return the message. It will be called only then the message actually gets logged, which is useful if it is costly to create the message in the first place.

This function takes care of saving and restoring `$!` and `$@`, so you don't have to.

Whether the given message will be logged depends on the maximum log level and the caller's package. The return value can be used to ensure that messages or not "lost" − for example, when AnyEvent::Debug detects a runtime error it tries to log it at `die` level, but if that message is lost it simply uses warn.

Note that you can (and should) call this function as `AnyEvent::log` or `AE::log`, without `use`−ing this module if possible (i.e. you don't need any additional functionality), as those functions will load the logging module on demand only. They are also much shorter to write.

Also, if you optionally generate a lot of debug messages (such as when tracing some code), you should look into using a logger callback and a boolean enabler (see `logger`, below).

Example: log something at error level.

```
AE::log error => "something";
```

Example: use printf-formatting.

```
AE::log info => "%5d %-10.10s %s", $index, $category, $msg;
```

Example: only generate a costly dump when the message is actually being logged.

```
AE::log debug => sub { require Data::Dump; Data::Dump::dump \%cache };
```

`$logger` = AnyEvent::Log::logger `$level`[, \$enabled]

Creates a code reference that, when called, acts as if the `AnyEvent::Log::log` function was called at this point with the given level. `$logger` is passed a `$msg` and optional `@args`, just as with the `AnyEvent::Log::log` function:

```
my $debug_log = AnyEvent::Log::logger "debug";

$debug_log->("debug here");
$debug_log->("%06d emails processed", 12345);
$debug_log->(sub { $obj->as_string });
```

The idea behind this function is to decide whether to log before actually logging − when the `logger` function is called once, but the returned logger callback often, then this can be a tremendous speed win.

Despite this speed advantage, changes in logging configuration will still be reflected by the logger callback, even if configuration changes *after* it was created.

To further speed up logging, you can bind a scalar variable to the logger, which contains true if the logger should be called or not − if it is false, calling the logger can be safely skipped. This variable will be updated as long as `$logger` is alive.

Full example:

```
   # near the init section
   use AnyEvent::Log;

   my $debug_log = AnyEvent:Log::logger debug => \my $debug;

   # and later in your program
   $debug_log->("yo, stuff here") if $debug;

   $debug and $debug_log->("123");
```

AnyEvent::Log::exact_time `$on`

By default, `AnyEvent::Log` will use `AE::now`, i.e. the cached eventloop time, for the log timestamps. After calling this function with a true value it will instead resort to `AE::time`, i.e. fetch the current time on each log message. This only makes a difference for event loops that actually cache the time (such as EV or AnyEvent::Loop).

This setting can be changed at any time by calling this function.

Since `AnyEvent::Log` has to work even before the AnyEvent has been initialised, this switch will also decide whether to use `CORE::time` or `Time::HiRes::time` when logging a message before AnyEvent becomes available.

AnyEvent::Log::format_time `$timestamp`

Formats a timestamp as returned by `AnyEvent->now` or `AnyEvent->time` or many other functions in the same way as `AnyEvent::Log` does.

In your main program (as opposed to in your module) you can override the default timestamp display format by loading this module and then redefining this function.

Most commonly, this function can be used in formatting callbacks.

AnyEvent::Log::default_format `$time`, `$ctx`, `$level`, `$msg`

Format a log message using the given timestamp, logging context, log level and log message.

This is the formatting function used to format messages when no custom function is provided.

In your main program (as opposed to in your module) you can override the default message format by loading this module and then redefining this function.

AnyEvent::Log::fatal_exit()

This is the function that is called after logging a `fatal` log message. It must not return.

The default implementation simply calls `exit 1`.

In your main program (as opposed to in your module) you can override the fatal exit function by loading this module and then redefining this function. Make sure you don't return.

LOGGING CONTEXTS

This module associates every log message with a so-called *logging context*, based on the package of the caller. Every perl package has its own logging context.

A logging context has three major responsibilities: filtering, logging and propagating the message.

For the first purpose, filtering, each context has a set of logging levels, called the log level mask. Messages not in the set will be ignored by this context (masked).

For logging, the context stores a formatting callback (which takes the timestamp, context, level and string message and formats it in the way it should be logged) and a logging callback (which is responsible for actually logging the formatted message and telling `AnyEvent::Log` whether it has consumed the message, or whether it should be propagated).

For propagation, a context can have any number of attached *slave contexts*. Any message that is neither masked by the logging mask nor masked by the logging callback returning true will be passed to all slave contexts.

Each call to a logging function will log the message at most once per context, so it does not matter (much) if there are cycles or if the message can arrive at the same context via multiple paths.

DEFAULTS

By default, all logging contexts have an full set of log levels ("all"), a disabled logging callback and the default formatting callback.

Package contexts have the package name as logging title by default.

They have exactly one slave – the context of the "parent" package. The parent package is simply defined to be the package name without the last component, i.e. `AnyEvent::Debug::Wrapped` becomes `AnyEvent::Debug`, and `AnyEvent` becomes ... `$AnyEvent::Log::COLLECT` which is the exception of the rule – just like the "parent" of any single-component package name in Perl is `main`, the default slave of any top-level package context is `$AnyEvent::Log::COLLECT`.

Since perl packages form only an approximate hierarchy, this slave context can of course be removed.

All other (anonymous) contexts have no slaves and an empty title by default.

When the module is loaded it creates the `$AnyEvent::Log::LOG` logging context that simply logs everything via `warn`, without propagating anything anywhere by default. The purpose of this context is to provide a convenient place to override the global logging target or to attach additional log targets. It's not meant for filtering.

It then creates the `$AnyEvent::Log::FILTER` context whose purpose is to suppress all messages with priority higher than `$ENV{PERL_ANYEVENT_VERBOSE}`. It then attached the `$AnyEvent::Log::LOG` context to it. The purpose of the filter context is to simply provide filtering according to some global log level.

Finally it creates the top-level package context `$AnyEvent::Log::COLLECT` and attaches the `$AnyEvent::Log::FILTER` context to it, but otherwise leaves it at default config. Its purpose is simply to collect all log messages system-wide.

The hierarchy is then:

```
   any package, eventually -> $COLLECT -> $FILTER -> $LOG
```

The effect of all this is that log messages, by default, wander up to the `$AnyEvent::Log::COLLECT` context where all messages normally end up, from there to `$AnyEvent::Log::FILTER` where log messages with lower priority then `$ENV{PERL_ANYEVENT_VERBOSE}` will be filtered out and then to the `$AnyEvent::Log::LOG` context to be passed to `warn`.

This makes it easy to set a global logging level (by modifying `$FILTER`), but still allow other contexts to send, for example, their debug and trace messages to the `$LOG` target despite the global logging level, or to attach additional log targets that log messages, regardless of the global logging level.

It also makes it easy to modify the default warn-logger ($LOG) to something that logs to a file, or to attach additional logging targets (such as loggign to a file) by attaching it to `$FILTER`.

CREATING/FINDING/DESTROYING CONTEXTS

`$ctx` = AnyEvent::Log::ctx [$pkg]

This function creates or returns a logging context (which is an object).

If a package name is given, then the context for that package is returned. If it is called without any arguments, then the context for the callers package is returned (i.e. the same context as a `AE::log` call would use).

If `undef` is given, then it creates a new anonymous context that is not tied to any package and is destroyed when no longer referenced.

AnyEvent::Log::reset

Resets all package contexts and recreates the default hierarchy if necessary, i.e. resets the logging subsystem to defaults, as much as possible. This process keeps references to contexts held by other parts of the program intact.

This can be used to implement config-file (re−)loading: before loading a configuration, reset all contexts.

`$ctx` = new AnyEvent::Log::Ctx methodname => param...

This is a convenience constructor that makes it simpler to construct anonymous logging contexts.

Each key-value pair results in an invocation of the method of the same name as the key with the value as parameter, unless the value is an arrayref, in which case it calls the method with the contents of the array. The methods are called in the same order as specified.

Example: create a new logging context and set both the default logging level, some slave contexts and a logging callback.

```
   $ctx = new AnyEvent::Log::Ctx
      title   => "dubious messages",
      level   => "error",
      log_cb  => sub { print STDOUT shift; 0 },
      slaves  => [$ctx1, $ctx, $ctx2],
   ;
```

CONFIGURING A LOG CONTEXT

The following methods can be used to configure the logging context.

`$ctx`−>title ([$new_title])

Returns the title of the logging context − this is the package name, for package contexts, and a user defined string for all others.

If `$new_title` is given, then it replaces the package name or title.

LOGGING LEVELS

The following methods deal with the logging level set associated with the log context.

The most common method to use is probably `$ctx->level ($level)`, which configures the specified and any higher priority levels.

All functions which accept a list of levels also accept the special string `all` which expands to all logging levels.

`$ctx`−>levels ($level[, `$level`...)

Enables logging for the given levels and disables it for all others.

`$ctx`−>level ($level)

Enables logging for the given level and all lower level (higher priority) ones. In addition to normal logging levels, specifying a level of `0` or `off` disables all logging for this level.

Example: log warnings, errors and higher priority messages.

```
   $ctx->level ("warn");
   $ctx->level (5); # same thing, just numeric
```

`$ctx`−>enable ($level[, `$level`...])

Enables logging for the given levels, leaving all others unchanged.

`$ctx`−>disable ($level[, `$level`...])

Disables logging for the given levels, leaving all others unchanged.

`$ctx->cap ($level)`

Caps the maximum priority to the given level, for all messages logged to, or passing through, this context. That is, while this doesn't affect whether a message is logged or passed on, the maximum priority of messages will be limited to the specified level − messages with a higher priority will be set to the specified priority.

Another way to view this is that `->level` filters out messages with a too low priority, while `->cap` modifies messages with a too high priority.

This is useful when different log targets have different interpretations of priority. For example, for a specific command line program, a wrong command line switch might well result in a `fatal` log message, while the same message, logged to syslog, is likely *not* fatal to the system or syslog facility as a whole, but more likely a mere `error`.

This can be modeled by having a stderr logger that logs messages "as-is" and a syslog logger that logs messages with a level cap of, say, `error`, or, for truly system-critical components, actually `critical`.

SLAVE CONTEXTS

The following methods attach and detach another logging context to a logging context.

Log messages are propagated to all slave contexts, unless the logging callback consumes the message.

`$ctx->attach ($ctx2[, $ctx3...])`

Attaches the given contexts as slaves to this context. It is not an error to add a context twice (the second add will be ignored).

A context can be specified either as package name or as a context object.

`$ctx->detach ($ctx2[, $ctx3...])`

Removes the given slaves from this context − it's not an error to attempt to remove a context that hasn't been added.

A context can be specified either as package name or as a context object.

`$ctx->slaves ($ctx2[, $ctx3...])`

Replaces all slaves attached to this context by the ones given.

LOG TARGETS

The following methods configure how the logging context actually does the logging (which consists of formatting the message and printing it or whatever it wants to do with it).

`$ctx->log_cb ($cb->($str))`

Replaces the logging callback on the context (`undef` disables the logging callback).

The logging callback is responsible for handling formatted log messages (see `fmt_cb` below) − normally simple text strings that end with a newline (and are possibly multiline themselves).

It also has to return true iff it has consumed the log message, and false if it hasn't. Consuming a message means that it will not be sent to any slave context. When in doubt, return `0` from your logging callback.

Example: a very simple logging callback, simply dump the message to STDOUT and do not consume it.

```
$ctx->log_cb (sub { print STDERR shift; 0 });
```

You can filter messages by having a log callback that simply returns `1` and does not do anything with the message, but this counts as "message being logged" and might not be very efficient.

Example: propagate all messages except for log levels "debug" and "trace". The messages will still be generated, though, which can slow down your program.

```
    $ctx->levels ("debug", "trace");
    $ctx->log_cb (sub { 1 }); # do not log, but eat debug and trace messages
```

$ctx->fmt_cb ($fmt_cb->($timestamp, $orig_ctx, $level, $message))

Replaces the formatting callback on the context (`undef` restores the default formatter).

The callback is passed the (possibly fractional) timestamp, the original logging context (object, not title), the (numeric) logging level and the raw message string and needs to return a formatted log message. In most cases this will be a string, but it could just as well be an array reference that just stores the values.

If, for some reason, you want to use `caller` to find out more about the logger then you should walk up the call stack until you are no longer inside the `AnyEvent::Log` package.

To implement your own logging callback, you might find the `AnyEvent::Log::format_time` and `AnyEvent::Log::default_format` functions useful.

Example: format the message just as AnyEvent::Log would, by letting AnyEvent::Log do the work. This is a good basis to design a formatting callback that only changes minor aspects of the formatting.

```
    $ctx->fmt_cb (sub {
       my ($time, $ctx, $lvl, $msg) = @_;

       AnyEvent::Log::default_format $time, $ctx, $lvl, $msg
    });
```

Example: format just the raw message, with numeric log level in angle brackets.

```
    $ctx->fmt_cb (sub {
       my ($time, $ctx, $lvl, $msg) = @_;

       "<$lvl>$msg\n"
    });
```

Example: return an array reference with just the log values, and use `PApp::SQL::sql_exec` to store the message in a database.

```
    $ctx->fmt_cb (sub { \@_ });
    $ctx->log_cb (sub {
       my ($msg) = @_;

       sql_exec "insert into log (when, subsys, prio, msg) values (?, ?, ?, ?)"
                $msg->[0] + 0,
                "$msg->[1]",
                $msg->[2] + 0,
                "$msg->[3]";

       0
    });
```

$ctx->log_to_warn

Sets the `log_cb` to simply use `CORE::warn` to report any messages (usually this logs to STDERR).

$ctx->log_to_file ($path)

Sets the `log_cb` to log to a file (by appending), unbuffered. The function might return before the log file has been opened or created.

$ctx->log_to_path ($path)

Same as `->log_to_file`, but opens the file for each message. This is much slower, but allows you to change/move/rename/delete the file at basically any time.

Needless(?) to say, if you do not want to be bitten by some evil person calling `chdir`, the path should be absolute. Doesn't help with `chroot`, but hey...

`$ctx`->log_to_syslog ([$facility])
: Logs all messages via Sys::Syslog, mapping `trace` to `debug` and all the others in the obvious way. If specified, then the `$facility` is used as the facility (`user`, `auth`, `local0` and so on). The default facility is `user`.

  Note that this function also sets a `fmt_cb` – the logging part requires an array reference with [$level, `$str`] as input.

MESSAGE LOGGING

These methods allow you to log messages directly to a context, without going via your package context.

`$ctx`->log ($level, `$msg`[, `@params`])
: Same as `AnyEvent::Log::log`, but uses the given context as log context.

  Example: log a message in the context of another package.

```
   (AnyEvent::Log::ctx "Other::Package")->log (warn => "heely bo");
```

`$logger` = `$ctx`->logger ($level[, \$enabled])
: Same as `AnyEvent::Log::logger`, but uses the given context as log context.

CONFIGURATION VIA `$ENV`{PERL_ANYEVENT_LOG}

Logging can also be configured by setting the environment variable `PERL_ANYEVENT_LOG` (or `AE_LOG`).

The value consists of one or more logging context specifications separated by `:` or whitespace. Each logging specification in turn starts with a context name, followed by =, followed by zero or more comma-separated configuration directives, here are some examples:

```
   # set default logging level
   filter=warn

   # log to file instead of to stderr
   log=file=/tmp/mylog

   # log to file in addition to stderr
   log=+%file:%file=file=/tmp/mylog

   # enable debug log messages, log warnings and above to syslog
   filter=debug:log=+%warnings:%warnings=warn,syslog=LOG_LOCAL0

   # log trace messages (only) from AnyEvent::Debug to file
   AnyEvent::Debug=+%trace:%trace=only,trace,file=/tmp/tracelog
```

A context name in the log specification can be any of the following:

`collect`, `filter`, `log`
: Correspond to the three predefined `$AnyEvent::Log::COLLECT`, `AnyEvent::Log::FILTER` and `$AnyEvent::Log::LOG` contexts.

`%name`
: Context names starting with a `%` are anonymous contexts created when the name is first mentioned. The difference to package contexts is that by default they have no attached slaves.

  This makes it possible to create new log contexts that can be referred to multiple times by name within the same log specification.

a perl package name
: Any other string references the logging context associated with the given Perl `package`. In the unlikely case where you want to specify a package context that matches on of the other context name forms, you can add a `::` to the package name to force interpretation as a package.

The configuration specifications can be any number of the following:

`stderr`
: Configures the context to use Perl's `warn` function (which typically logs to `STDERR`). Works like `log_to_warn`.

`file=`*path*
: Configures the context to log to a file with the given path. Works like `log_to_file`.

`path=`*path*
: Configures the context to log to a file with the given path. Works like `log_to_path`.

`syslog` or `syslog=`*expr*
: Configures the context to log to syslog. If *expr* is given, then it is evaluated in the Sys::Syslog package, so you could use:

```
    log=syslog=LOG_LOCAL0
```

`nolog`
: Configures the context to not log anything by itself, which is the default. Same as `$ctx->log_cb (undef)`.

`cap=`*level*
: Caps logging messages entering this context at the given level, i.e. reduces the priority of messages with higher priority than this level. The default is `0` (or `off`), meaning the priority will not be touched.

`0` or `off`
: Sets the logging level of the context to `0`, i.e. all messages will be filtered out.

`all`
: Enables all logging levels, i.e. filtering will effectively be switched off (the default).

`only`
: Disables all logging levels, and changes the interpretation of following level specifications to enable the specified level only.

Example: only enable debug messages for a context.

```
    context=only,debug
```

`except`
: Enables all logging levels, and changes the interpretation of following level specifications to disable that level. Rarely used.

Example: enable all logging levels except fatal and trace (this is rather nonsensical).

```
    filter=exept,fatal,trace
```

`level`
: Enables all logging levels, and changes the interpretation of following level specifications to be "that level or any higher priority message". This is the default.

Example: log anything at or above warn level.

```
    filter=warn

    # or, more verbose
    filter=only,level,warn
```

`1..9` or a logging level name (`error`, `debug` etc.)
: A numeric loglevel or the name of a loglevel will be interpreted according to the most recent `only`, `except` or `level` directive. By default, specifying a logging level enables that and any higher priority messages.

+context
: Attaches the named context as slave to the context.

+
: A lone + detaches all contexts, i.e. clears the slave list from the context. Anonymous (`%name`) contexts have no attached slaves by default, but package contexts have the parent context as slave by default.

  Example: log messages from My::Module to a file, do not send them to the default log collector.

```
My::Module=+,file=/tmp/mymodulelog
```

Any character can be escaped by prefixing it with a \ (backslash), as usual, so to log to a file containing a comma, colon, backslash and some spaces in the filename, you would do this:

```
PERL_ANYEVENT_LOG='log=file=/some\ \:file\ with\,\ \\-escapes'
```

Since whitespace (which includes newlines) is allowed, it is fine to specify multiple lines in `PERL_ANYEVENT_LOG`, e.g.:

```
PERL_ANYEVENT_LOG="
   filter=warn
   AnyEvent::Debug=+%trace
   %trace=only,trace,+log
" myprog
```

Also, in the unlikely case when you want to concatenate specifications, use whitespace as separator, as `::` will be interpreted as part of a module name, an empty spec with two separators:

```
PERL_ANYEVENT_LOG="$PERL_ANYEVENT_LOG MyMod=debug"
```

EXAMPLES

This section shows some common configurations, both as code, and as `PERL_ANYEVENT_LOG` string.

Setting the global logging level.
: Either put `PERL_ANYEVENT_VERBOSE=`<number> into your environment before running your program, use `PERL_ANYEVENT_LOG` or modify the log level of the root context at runtime:

```
PERL_ANYEVENT_VERBOSE=5 ./myprog

PERL_ANYEVENT_LOG=log=warn

$AnyEvent::Log::FILTER->level ("warn");
```

Append all messages to a file instead of sending them to STDERR.
: This is affected by the global logging level.

```
$AnyEvent::Log::LOG->log_to_file ($path);

PERL_ANYEVENT_LOG=log=file=/some/path
```

Write all messages with priority `error` and higher to a file.
: This writes them only when the global logging level allows it, because it is attached to the default context which is invoked *after* global filtering.

```
$AnyEvent::Log::FILTER->attach (
   new AnyEvent::Log::Ctx log_to_file => $path);

PERL_ANYEVENT_LOG=filter=+%filelogger:%filelogger=file=/some/path
```

  This writes them regardless of the global logging level, because it is attached to the toplevel context, which receives all messages *before* the global filtering.

```
$AnyEvent::Log::COLLECT->attach (
   new AnyEvent::Log::Ctx log_to_file => $path);
```

```
PERL_ANYEVENT_LOG=%filelogger=file=/some/path:collect=+%filelogger
```

In both cases, messages are still written to STDERR.

Additionally log all messages with `warn` and higher priority to `syslog`, but cap at `error`.
This logs all messages to the default log target, but also logs messages with priority `warn` or higher (and not filtered otherwise) to syslog facility `user`. Messages with priority higher than `error` will be logged with level `error`.

```
$AnyEvent::Log::LOG->attach (
   new AnyEvent::Log::Ctx
      level  => "warn",
      cap    => "error",
      syslog => "user",
);

PERL_ANYEVENT_LOG=log=+%syslog:%syslog=warn,cap=error,syslog
```

Write trace messages (only) from AnyEvent::Debug to the default logging target(s).
Attach the `$AnyEvent::Log::LOG` context to the `AnyEvent::Debug` context − this simply circumvents the global filtering for trace messages.

```
my $debug = AnyEvent::Debug->AnyEvent::Log::ctx;
$debug->attach ($AnyEvent::Log::LOG);

PERL_ANYEVENT_LOG=AnyEvent::Debug=+log
```

This of course works for any package, not just AnyEvent::Debug, but assumes the log level for AnyEvent::Debug hasn't been changed from the default.

ASYNCHRONOUS DISK I/O

This module uses AnyEvent::IO to actually write log messages (in `log_to_file` and `log_to_path`), so it doesn't block your program when the disk is busy and a non-blocking AnyEvent::IO backend is available.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```