NAME

EV − perl interface to libev, a high performance full−featured event loop

SYNOPSIS

```
use EV;

# TIMERS

my $w = EV::timer 2, 0, sub {
   warn "is called after 2s";
};

my $w = EV::timer 2, 2, sub {
   warn "is called roughly every 2s (repeat = 2)";
};

undef $w; # destroy event watcher again

my $w = EV::periodic 0, 60, 0, sub {
   warn "is called every minute, on the minute, exactly";
};

# IO

my $w = EV::io *STDIN, EV::READ, sub {
   my ($w, $revents) = @_; # all callbacks receive the watcher and event mask
   warn "stdin is readable, you entered: ", <STDIN>;
};

# SIGNALS

my $w = EV::signal 'QUIT', sub {
   warn "sigquit received\n";
};

# CHILD/PID STATUS CHANGES

my $w = EV::child 666, 0, sub {
   my ($w, $revents) = @_;
   my $status = $w->rstatus;
};

# STAT CHANGES
my $w = EV::stat "/etc/passwd", 10, sub {
   my ($w, $revents) = @_;
   warn $w->path, " has changed somehow.\n";
};

# MAINLOOP
EV::run;                # loop until EV::break is called or all watchers stop
EV::run EV::RUN_ONCE;   # block until at least one event could be handled
EV::run EV::RUN_NOWAIT; # try to handle same events, but do not block
```

BEFORE YOU START USING THIS MODULE

If you only need timer, I/O, signal, child and idle watchers and not the advanced functionality of this module, consider using AnyEvent instead, specifically the simplified API described in AE.

When used with EV as backend, the AE API is as fast as the native EV API, but your programs/modules will still run with many other event loops.

DESCRIPTION

This module provides an interface to libev (<http://software.schmorp.de/pkg/libev.html>). While the documentation below is comprehensive, one might also consult the documentation of libev itself (<http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod> or *perldoc EV::libev*) for more subtle details on watcher semantics or some discussion on the available backends, or how to force a specific backend with `LIBEV_FLAGS`, or just about in any case because it has much more detailed information.

This module is very fast and scalable. It is actually so fast that you can use it through the AnyEvent module, stay portable to other event loops (if you don't rely on any watcher types not available through it) and still be faster than with any other event loop currently supported in Perl.

PORTING FROM EV 3.X to 4.X

EV version 4 introduces a number of incompatible changes summarised here. According to the depreciation strategy used by libev, there is a compatibility layer in place so programs should continue to run unchanged (the XS interface lacks this layer, so programs using that one need to be updated).

This compatibility layer will be switched off in some future release.

All changes relevant to Perl are renames of symbols, functions and methods:

```
EV::loop           => EV::run
EV::LOOP_NONBLOCK => EV::RUN_NOWAIT
EV::LOOP_ONESHOT  => EV::RUN_ONCE

EV::unloop         => EV::break
EV::UNLOOP_CANCEL => EV::BREAK_CANCEL
EV::UNLOOP_ONE     => EV::BREAK_ONE
EV::UNLOOP_ALL     => EV::BREAK_ALL

EV::TIMEOUT        => EV::TIMER

EV::loop_count     => EV::iteration
EV::loop_depth     => EV::depth
EV::loop_verify    => EV::verify
```

The loop object methods corresponding to the functions above have been similarly renamed.

MODULE EXPORTS

This module does not export any symbols.

EVENT LOOPS

EV supports multiple event loops: There is a single ''default event loop'' that can handle everything including signals and child watchers, and any number of ''dynamic event loops'' that can use different backends (with various limitations), but no child and signal watchers.

You do not have to do anything to create the default event loop: When the module is loaded a suitable backend is selected on the premise of selecting a working backend (which for example rules out kqueue on most BSDs). Modules should, unless they have ''special needs'' always use the default loop as this is fastest (perl-wise), best supported by other modules (e.g. AnyEvent or Coro) and most portable event loop.

For specific programs you can create additional event loops dynamically.

If you want to take advantage of kqueue (which often works properly for sockets only) even though the default loop doesn't enable it, you can *embed* a kqueue loop into the default loop: running the default loop will then also service the kqueue loop to some extent. See the example in the section about embed watchers for an example on how to achieve that.

`$loop` = new EV::Loop [$flags]
: Create a new event loop as per the specified flags. Please refer to the `ev_loop_new ()` function description in the libev documentation (<http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#GLOBAL_FUNCTIONS>, or locally-installed as *EV::libev* manpage) for more info.

  The loop will automatically be destroyed when it is no longer referenced by any watcher and the loop object goes out of scope.

  If you are not embedding the loop, then Using `EV::FLAG_FORKCHECK` is recommended, as only the default event loop is protected by this module. If you *are* embedding this loop in the default loop, this is not necessary, as `EV::embed` automatically does the right thing on fork.

`$loop->loop_fork`
: Must be called after a fork in the child, before entering or continuing the event loop. An alternative is to use `EV::FLAG_FORKCHECK` which calls this function automatically, at some performance loss (refer to the libev documentation).

`$loop->verify`
: Calls `ev_verify` to make internal consistency checks (for debugging libev) and abort the program if any data structures were found to be corrupted.

`$loop` = EV::default_loop [$flags]
: Return the default loop (which is a singleton object). Since this module already creates the default loop with default flags, specifying flags here will not have any effect unless you destroy the default loop first, which isn't supported. So in short: don't do it, and if you break it, you get to keep the pieces.

BASIC INTERFACE

`$EV::DIED`
: Must contain a reference to a function that is called when a callback throws an exception (with $@ containing the error). The default prints an informative message and continues.

  If this callback throws an exception it will be silently ignored.

`$flags` = EV::supported_backends
`$flags` = EV::recommended_backends
`$flags` = EV::embeddable_backends
: Returns the set (see `EV::BACKEND_*` flags) of backends supported by this instance of EV, the set of recommended backends (supposed to be good) for this platform and the set of embeddable backends (see EMBED WATCHERS).

EV::sleep `$seconds`
: Block the process for the given number of (fractional) seconds.

`$time` = EV::time
: Returns the current time in (fractional) seconds since the epoch.

`$time` = EV::now
`$time` = `$loop->now`
: Returns the time the last event loop iteration has been started. This is the time that (relative) timers are based on, and referring to it is usually faster then calling EV::time.

EV::now_update
`$loop->now_update`
: Establishes the current time by querying the kernel, updating the time returned by `EV::now` in the progress. This is a costly operation and is usually done automatically within `EV::run`.

  This function is rarely useful, but when some event callback runs for a very long time without entering the event loop, updating libev's idea of the current time is a good idea.

EV::suspend
$loop->suspend
EV::resume
$loop->resume
These two functions suspend and resume a loop, for use when the loop is not used for a while and timeouts should not be processed.

A typical use case would be an interactive program such as a game: When the user presses ^Z to suspend the game and resumes it an hour later it would be best to handle timeouts as if no time had actually passed while the program was suspended. This can be achieved by calling suspend in your SIGTSTP handler, sending yourself a SIGSTOP and calling resume directly afterwards to resume timer processing.

Effectively, all timer watchers will be delayed by the time spend between suspend and resume, and all periodic watchers will be rescheduled (that is, they will lose any events that would have occurred while suspended).

After calling suspend you **must not** call *any* function on the given loop other than resume, and you **must not** call resume without a previous call to suspend.

Calling suspend/resume has the side effect of updating the event loop time (see now_update).

$backend = EV::backend
$backend = $loop->backend
Returns an integer describing the backend used by libev (EV::BACKEND_SELECT or EV::BACKEND_EPOLL).

$active = EV::run [$flags]
$active = $loop->run ([$flags])
Begin checking for events and calling callbacks. It returns when a callback calls EV::break or the flags are nonzero (in which case the return value is true) or when there are no active watchers which reference the loop (keepalive is true), in which case the return value will be false. The return value can generally be interpreted as ''if true, there is more work left to do''.

The $flags argument can be one of the following:

```
0               as above
EV::RUN_ONCE    block at most once (wait, but do not loop)
EV::RUN_NOWAIT  do not block at all (fetch/handle events but do not wait)
```

EV::break [$how]
$loop->break ([$how])
When called with no arguments or an argument of EV::BREAK_ONE, makes the innermost call to EV::run return.

When called with an argument of EV::BREAK_ALL, all calls to EV::run will return as fast as possible.

When called with an argument of EV::BREAK_CANCEL, any pending break will be cancelled.

$count = EV::iteration
$count = $loop->iteration
Return the number of times the event loop has polled for new events. Sometimes useful as a generation counter.

EV::once $fh_or_undef, $events, $timeout, $cb->($revents)
$loop->once ($fh_or_undef, $events, $timeout, $cb->($revents))
This function rolls together an I/O and a timer watcher for a single one-shot event without the need for managing a watcher object.

If $fh_or_undef is a filehandle or file descriptor, then $events must be a bitset containing either EV::READ, EV::WRITE or EV::READ | EV::WRITE, indicating the type of I/O event you want to wait for. If you do not want to wait for some I/O event, specify undef for $fh_or_undef and 0

for `$events`).

If timeout is `undef` or negative, then there will be no timeout. Otherwise an `EV::timer` with this value will be started.

When an error occurs or either the timeout or I/O watcher triggers, then the callback will be called with the received event set (in general you can expect it to be a combination of `EV::ERROR`, `EV::READ`, `EV::WRITE` and `EV::TIMER`).

EV::once doesn't return anything: the watchers stay active till either of them triggers, then they will be stopped and freed, and the callback invoked.

EV::feed_fd_event `$fd`, `$revents`
`$loop->`feed_fd_event ($fd, `$revents`)
Feed an event on a file descriptor into EV. EV will react to this call as if the readyness notifications specified by `$revents` (a combination of `EV::READ` and `EV::WRITE`) happened on the file descriptor `$fd`.

EV::feed_signal_event `$signal`
Feed a signal event into the default loop. EV will react to this call as if the signal specified by `$signal` had occurred.

EV::feed_signal `$signal`
Feed a signal event into EV - unlike `EV::feed_signal_event`, this works regardless of which loop has registered the signal, and is mainly useful fro custom signal implementations.

EV::set_io_collect_interval `$time`
`$loop->`set_io_collect_interval ($time)
EV::set_timeout_collect_interval `$time`
`$loop->`set_timeout_collect_interval ($time)
These advanced functions set the minimum block interval when polling for I/O events and the minimum wait interval for timer events. See the libev documentation at <http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#FUNCTIONS_CONTROLLING_THE_EVENT_LOOP> (locally installed as *EV::libev*) for a more detailed discussion.

`$count` = EV::pending_count
`$count` = `$loop->`pending_count
Returns the number of currently pending watchers.

EV::invoke_pending
`$loop->`invoke_pending
Invoke all currently pending watchers.

WATCHER OBJECTS

A watcher is an object that gets created to record your interest in some event. For instance, if you want to wait for STDIN to become readable, you would create an EV::io watcher for that:

```
my $watcher = EV::io *STDIN, EV::READ, sub {
   my ($watcher, $revents) = @_;
   warn "yeah, STDIN should now be readable without blocking!\n"
};
```

All watchers can be active (waiting for events) or inactive (paused). Only active watchers will have their callbacks invoked. All callbacks will be called with at least two arguments: the watcher and a bitmask of received events.

Each watcher type has its associated bit in revents, so you can use the same callback for multiple watchers. The event mask is named after the type, i.e. EV::child sets EV::CHILD, EV::prepare sets EV::PREPARE, EV::periodic sets EV::PERIODIC and so on, with the exception of I/O events (which can set both EV::READ and EV::WRITE bits).

In the rare case where one wants to create a watcher but not start it at the same time, each constructor has a variant with a trailing `_ns` in its name, e.g. EV::io has a non-starting variant EV::io_ns and so on.

Please note that a watcher will automatically be stopped when the watcher object is destroyed, so you *need* to keep the watcher objects returned by the constructors.

Also, all methods changing some aspect of a watcher (->set, ->priority, ->fh and so on) automatically stop and start it again if it is active, which means pending events get lost.

COMMON WATCHER METHODS

This section lists methods common to all watchers.

`$w->start`
: Starts a watcher if it isn't active already. Does nothing to an already active watcher. By default, all watchers start out in the active state (see the description of the `_ns` variants if you need stopped watchers).

`$w->stop`
: Stop a watcher if it is active. Also clear any pending events (events that have been received but that didn't yet result in a callback invocation), regardless of whether the watcher was active or not.

`$bool = $w->is_active`
: Returns true if the watcher is active, false otherwise.

`$current_data = $w->data`
`$old_data = $w->data ($new_data)`
: Queries a freely usable data scalar on the watcher and optionally changes it. This is a way to associate custom data with a watcher:

```
my $w = EV::timer 60, 0, sub {
   warn $_[0]->data;
};
$w->data ("print me!");
```

`$current_cb = $w->cb`
`$old_cb = $w->cb ($new_cb)`
: Queries the callback on the watcher and optionally changes it. You can do this at any time without the watcher restarting.

`$current_priority = $w->priority`
`$old_priority = $w->priority ($new_priority)`
: Queries the priority on the watcher and optionally changes it. Pending watchers with higher priority will be invoked first. The valid range of priorities lies between EV::MAXPRI (default 2) and EV::MINPRI (default -2). If the priority is outside this range it will automatically be normalised to the nearest valid priority.

  The default priority of any newly-created watcher is 0.

  Note that the priority semantics have not yet been fleshed out and are subject to almost certain change.

`$w->invoke ($revents)`
: Call the callback *now* with the given event mask.

`$w->feed_event ($revents)`
: Feed some events on this watcher into EV. EV will react to this call as if the watcher had received the given `$revents` mask.

`$revents = $w->clear_pending`
: If the watcher is pending, this function clears its pending status and returns its `$revents` bitset (as if its callback was invoked). If the watcher isn't pending it does nothing and returns `0`.

`$previous_state = $w->keepalive ($bool)`
: Normally, `EV::run` will return when there are no active watchers (which is a "deadlock" because no progress can be made anymore). This is convenient because it allows you to start your watchers (and your jobs), call `EV::run` once and when it returns you know that all your jobs are finished (or they forgot to register some watchers for their task :).

Sometimes, however, this gets in your way, for example when the module that calls `EV::run` (usually the main program) is not the same module as a long-living watcher (for example a DNS client module written by somebody else even). Then you might want any outstanding requests to be handled, but you would not want to keep `EV::run` from returning just because you happen to have this long-running UDP port watcher.

In this case you can clear the keepalive status, which means that even though your watcher is active, it won't keep `EV::run` from returning.

The initial value for keepalive is true (enabled), and you can change it any time.

Example: Register an I/O watcher for some UDP socket but do not keep the event loop from running just because of that watcher.

```
my $udp_socket = ...
my $udp_watcher = EV::io $udp_socket, EV::READ, sub { ... };
$udp_watcher->keepalive (0);
```

`$loop` = `$w->loop`

Return the loop that this watcher is attached to.

WATCHER TYPES

Each of the following subsections describes a single watcher type.

I/O WATCHERS − is this file descriptor readable or writable?

`$w` = EV::io `$fileno_or_fh`, `$eventmask`, `$callback`
`$w` = EV::io_ns `$fileno_or_fh`, `$eventmask`, `$callback`
`$w` = `$loop->io` ($fileno_or_fh, `$eventmask`, `$callback`)
`$w` = `$loop->io_ns` ($fileno_or_fh, `$eventmask`, `$callback`)

As long as the returned watcher object is alive, call the `$callback` when at least one of events specified in `$eventmask` occurs.

The `$eventmask` can be one or more of these constants ORed together:

```
EV::READ     wait until read() wouldn't block anymore
EV::WRITE    wait until write() wouldn't block anymore
```

The `io_ns` variant doesn't start (activate) the newly created watcher.

`$w->set` ($fileno_or_fh, `$eventmask`)

Reconfigures the watcher, see the constructor above for details. Can be called at any time.

`$current_fh` = `$w->fh`
`$old_fh` = `$w->fh` ($new_fh)

Returns the previously set filehandle and optionally set a new one.

`$current_eventmask` = `$w->events`
`$old_eventmask` = `$w->events` ($new_eventmask)

Returns the previously set event mask and optionally set a new one.

TIMER WATCHERS − relative and optionally repeating timeouts

`$w` = EV::timer `$after`, `$repeat`, `$callback`
`$w` = EV::timer_ns `$after`, `$repeat`, `$callback`
`$w` = `$loop->timer` ($after, `$repeat`, `$callback`)
`$w` = `$loop->timer_ns` ($after, `$repeat`, `$callback`)

Calls the callback after `$after` seconds (which may be fractional or negative). If `$repeat` is non-zero, the timer will be restarted (with the `$repeat` value as `$after`) after the callback returns.

This means that the callback would be called roughly after `$after` seconds, and then every `$repeat` seconds. The timer does his best not to drift, but it will not invoke the timer more often then once per event loop iteration, and might drift in other cases. If that isn't acceptable, look at EV::periodic, which can provide long-term stable timers.

The timer is based on a monotonic clock, that is, if somebody is sitting in front of the machine while the timer is running and changes the system clock, the timer will nevertheless run (roughly) the same time.

The `timer_ns` variant doesn't start (activate) the newly created watcher.

`$w->`set ($after, `$repeat` = 0)
Reconfigures the watcher, see the constructor above for details. Can be called at any time.

`$w->`again
`$w->`again ($repeat)
Similar to the `start` method, but has special semantics for repeating timers:

If the timer is active and non-repeating, it will be stopped.

If the timer is active and repeating, reset the timeout to occur `$repeat` seconds after now.

If the timer is inactive and repeating, start it using the repeat value.

Otherwise do nothing.

This behaviour is useful when you have a timeout for some IO operation. You create a timer object with the same value for `$after` and `$repeat`, and then, in the read/write watcher, run the `again` method on the timeout.

If called with a `$repeat` argument, then it uses this a timer repeat value.

`$after` = `$w->`remaining
Calculates and returns the remaining time till the timer will fire.

PERIODIC WATCHERS − to cron or not to cron?

`$w` = EV::periodic `$at`, `$interval`, `$reschedule_cb`, `$callback`
`$w` = EV::periodic_ns `$at`, `$interval`, `$reschedule_cb`, `$callback`
`$w` = `$loop->`periodic ($at, `$interval`, `$reschedule_cb`, `$callback`)
`$w` = `$loop->`periodic_ns ($at, `$interval`, `$reschedule_cb`, `$callback`)
Similar to EV::timer, but is not based on relative timeouts but on absolute times. Apart from creating "simple" timers that trigger "at" the specified time, it can also be used for non-drifting absolute timers and more complex, cron-like, setups that are not adversely affected by time jumps (i.e. when the system clock is changed by explicit date −s or other means such as ntpd). It is also the most complex watcher type in EV.

It has three distinct "modes":

- absolute timer ($interval = `$reschedule_cb` = 0)

  This time simply fires at the wallclock time `$at` and doesn't repeat. It will not adjust when a time jump occurs, that is, if it is to be run at January 1st 2011 then it will run when the system time reaches or surpasses this time.

- repeating interval timer ($interval > 0, `$reschedule_cb` = 0)

  In this mode the watcher will always be scheduled to time out at the next `$at + N * $interval` time (for the lowest integer N) and then repeat, regardless of any time jumps. Note that, since `N` can be negative, the first trigger can happen before `$at`.

  This can be used to create timers that do not drift with respect to system time:

```
my $hourly = EV::periodic 0, 3600, 0, sub { print "once/hour\n" };
```

  That doesn't mean there will always be 3600 seconds in between triggers, but only that the the callback will be called when the system time shows a full hour (UTC).

  Another way to think about it (for the mathematically inclined) is that EV::periodic will try to run the callback in this mode at the next possible time where `$time = $at (mod $interval)`, regardless of any time jumps.

- manual reschedule mode ($reschedule_cb = coderef)

  In this mode `$interval` and `$at` are both being ignored. Instead, each time the periodic watcher gets scheduled, the reschedule callback ($reschedule_cb) will be called with the watcher as first, and the current time as second argument.

  This callback MUST NOT stop or destroy this or any other periodic watcher, ever, and MUST NOT call any event loop functions or methods. If you need to stop it, return 1e30 and stop it afterwards. You may create and start an `EV::prepare` watcher for this task.

  It must return the next time to trigger, based on the passed time value (that is, the lowest time value larger than or equal to to the second argument). It will usually be called just before the callback will be triggered, but might be called at other times, too.

  This can be used to create very complex timers, such as a timer that triggers on each midnight, local time (actually one day after the last midnight, to keep the example simple):

```
my $daily = EV::periodic 0, 0, sub {
   my ($w, $now) = @_;

   use Time::Local ();
   my (undef, undef, undef, $d, $m, $y) = localtime $now;
   Time::Local::timelocal_nocheck 0, 0, 0, $d + 1, $m, $y
}, sub {
   print "it's midnight or likely shortly after, now\n";
};
```

The `periodic_ns` variant doesn't start (activate) the newly created watcher.

`$w->set` ($at, `$interval`, `$reschedule_cb`)
Reconfigures the watcher, see the constructor above for details. Can be called at any time.

`$w->again`
Simply stops and starts the watcher again.

`$time` = `$w->at`
Return the time that the watcher is expected to trigger next.

SIGNAL WATCHERS - signal me when a signal gets signalled!

`$w` = EV::signal `$signal`, `$callback`
`$w` = EV::signal_ns `$signal`, `$callback`
`$w` = `$loop->signal` ($signal, `$callback`)
`$w` = `$loop->signal_ns` ($signal, `$callback`)
Call the callback when `$signal` is received (the signal can be specified by number or by name, just as with `kill` or `%SIG`).

Only one event loop can grab a given signal - attempting to grab the same signal from two EV loops will crash the program immediately or cause data corruption.

EV will grab the signal for the process (the kernel only allows one component to receive a signal at a time) when you start a signal watcher, and removes it again when you stop it. Perl does the same when you add/remove callbacks to `%SIG`, so watch out.

You can have as many signal watchers per signal as you want.

The `signal_ns` variant doesn't start (activate) the newly created watcher.

`$w->set` ($signal)
Reconfigures the watcher, see the constructor above for details. Can be called at any time.

`$current_signum` = `$w->signal`

`$old_signum` = `$w->`signal ($new_signal)
Returns the previously set signal (always as a number not name) and optionally set a new one.

CHILD WATCHERS − watch out for process status changes

`$w` = EV::child `$pid`, `$trace`, `$callback`
`$w` = EV::child_ns `$pid`, `$trace`, `$callback`
`$w` = `$loop->`child ($pid, `$trace`, `$callback`)
`$w` = `$loop->`child_ns ($pid, `$trace`, `$callback`)
Call the callback when a status change for pid `$pid` (or any pid if `$pid` is 0) has been received (a status change happens when the process terminates or is killed, or, when trace is true, additionally when it is stopped or continued). More precisely: when the process receives a `SIGCHLD`, EV will fetch the outstanding exit/wait status for all changed/zombie children and call the callback.

It is valid (and fully supported) to install a child watcher after a child has exited but before the event loop has started its next iteration (for example, first you `fork`, then the new child process might exit, and only then do you install a child watcher in the parent for the new pid).

You can access both exit (or tracing) status and pid by using the `rstatus` and `rpid` methods on the watcher object.

You can have as many pid watchers per pid as you want, they will all be called.

The `child_ns` variant doesn't start (activate) the newly created watcher.

`$w->`set ($pid, `$trace`)
Reconfigures the watcher, see the constructor above for details. Can be called at any time.

`$current_pid` = `$w->`pid
Returns the previously set process id and optionally set a new one.

`$exit_status` = `$w->`rstatus
Return the exit/wait status (as returned by waitpid, see the waitpid entry in perlfunc).

`$pid` = `$w->`rpid
Return the pid of the awaited child (useful when you have installed a watcher for all pids).

STAT WATCHERS − did the file attributes just change?

`$w` = EV::stat `$path`, `$interval`, `$callback`
`$w` = EV::stat_ns `$path`, `$interval`, `$callback`
`$w` = `$loop->`stat ($path, `$interval`, `$callback`)
`$w` = `$loop->`stat_ns ($path, `$interval`, `$callback`)
Call the callback when a file status change has been detected on `$path`. The `$path` does not need to exist, changing from ''path exists'' to ''path does not exist'' is a status change like any other.

The `$interval` is a recommended polling interval for systems where OS-supported change notifications don't exist or are not supported. If you use `0` then an unspecified default is used (which is highly recommended!), which is to be expected to be around five seconds usually.

This watcher type is not meant for massive numbers of stat watchers, as even with OS-supported change notifications, this can be resource-intensive.

The `stat_ns` variant doesn't start (activate) the newly created watcher.

... = `$w->`stat
This call is very similar to the perl `stat` built-in: It stats (using `lstat`) the path specified in the watcher and sets perls stat cache (as well as EV's idea of the current stat values) to the values found.

In scalar context, a boolean is return indicating success or failure of the stat. In list context, the same 13−value list as with stat is returned (except that the blksize and blocks fields are not reliable).

In the case of an error, errno is set to `ENOENT` (regardless of the actual error value) and the `nlink` value is forced to zero (if the stat was successful then nlink is guaranteed to be non-zero).

See also the next two entries for more info.

... = $w->attr
Just like `$w->stat`, but without the initial stat'ing: this returns the values most recently detected by EV. See the next entry for more info.

... = $w->prev
Just like `$w->stat`, but without the initial stat'ing: this returns the previous set of values, before the change.

That is, when the watcher callback is invoked, `$w->prev` will be set to the values found *before* a change was detected, while `$w->attr` returns the values found leading to the change detection. The difference (if any) between `prev` and `attr` is what triggered the callback.

If you did something to the filesystem object and do not want to trigger yet another change, you can call `stat` to update EV's idea of what the current attributes are.

$w->set ($path, `$interval`)
Reconfigures the watcher, see the constructor above for details. Can be called at any time.

`$current_path` = $w->path
`$old_path` = $w->path ($new_path)
Returns the previously set path and optionally set a new one.

`$current_interval` = $w->interval
`$old_interval` = $w->interval ($new_interval)
Returns the previously set interval and optionally set a new one. Can be used to query the actual interval used.

IDLE WATCHERS - when you've got nothing better to do...

$w = EV::idle `$callback`
$w = EV::idle_ns `$callback`
$w = `$loop`->idle ($callback)
$w = `$loop`->idle_ns ($callback)
Call the callback when there are no other pending watchers of the same or higher priority (excluding check, prepare and other idle watchers of the same or lower priority, of course). They are called idle watchers because when the watcher is the highest priority pending event in the process, the process is considered to be idle at that priority.

If you want a watcher that is only ever called when *no* other events are outstanding you have to set the priority to `EV::MINPRI`.

The process will not block as long as any idle watchers are active, and they will be called repeatedly until stopped.

For example, if you have idle watchers at priority `0` and `1`, and an I/O watcher at priority `0`, then the idle watcher at priority `1` and the I/O watcher will always run when ready. Only when the idle watcher at priority `1` is stopped and the I/O watcher at priority `0` is not pending with the `0`-priority idle watcher be invoked.

The `idle_ns` variant doesn't start (activate) the newly created watcher.

PREPARE WATCHERS - customise your event loop!

$w = EV::prepare `$callback`
$w = EV::prepare_ns `$callback`
$w = `$loop`->prepare ($callback)
$w = `$loop`->prepare_ns ($callback)
Call the callback just before the process would block. You can still create/modify any watchers at this point.

See the EV::check watcher, below, for explanations and an example.

The `prepare_ns` variant doesn't start (activate) the newly created watcher.

CHECK WATCHERS − customise your event loop even more!

`$w` = EV::check `$callback`
`$w` = EV::check_ns `$callback`
`$w` = `$loop`->check ($callback)
`$w` = `$loop`->check_ns ($callback)

Call the callback just after the process wakes up again (after it has gathered events), but before any other callbacks have been invoked.

This can be used to integrate other event-based software into the EV mainloop: You register a prepare callback and in there, you create io and timer watchers as required by the other software. Here is a real-world example of integrating Net::SNMP (with some details left out):

```
our @snmp_watcher;

our $snmp_prepare = EV::prepare sub {
   # do nothing unless active
   $dispatcher->{_event_queue_h}
      or return;

   # make the dispatcher handle any outstanding stuff
   ... not shown

   # create an I/O watcher for each and every socket
   @snmp_watcher = (
      (map { EV::io $_, EV::READ, sub { } }
          keys %{ $dispatcher->{_descriptors} }),

      EV::timer +($event->[Net::SNMP::Dispatcher::_ACTIVE]
                  ? $event->[Net::SNMP::Dispatcher::_TIME] - EV::now : 0),
                0, sub { },
   );
};
```

The callbacks are irrelevant (and are not even being called), the only purpose of those watchers is to wake up the process as soon as one of those events occurs (socket readable, or timer timed out). The corresponding EV::check watcher will then clean up:

```
our $snmp_check = EV::check sub {
   # destroy all watchers
   @snmp_watcher = ();

   # make the dispatcher handle any new stuff
   ... not shown
};
```

The callbacks of the created watchers will not be called as the watchers are destroyed before this can happen (remember EV::check gets called first).

The `check_ns` variant doesn't start (activate) the newly created watcher.

EV::CHECK constant issues

Like all other watcher types, there is a bitmask constant for use in `$revents` and other places. The `EV::CHECK` is special as it has the same name as the `CHECK` sub called by Perl. This doesn't cause big issues on newer perls (beginning with 5.8.9), but it means thatthe constant must be *inlined*, i.e. runtime calls will not work. That means that as long as you always `use EV` and then `EV::CHECK` you are on the safe side.

FORK WATCHERS − the audacity to resume the event loop after a fork

Fork watchers are called when a `fork ()` was detected. The invocation is done before the event loop blocks next and before `check` watchers are being called, and only in the child after the fork.

`$w` = EV::fork `$callback`
`$w` = EV::fork_ns `$callback`
`$w` = `$loop`−>fork ($callback)
`$w` = `$loop`−>fork_ns ($callback)

Call the callback before the event loop is resumed in the child process after a fork.

The `fork_ns` variant doesn't start (activate) the newly created watcher.

EMBED WATCHERS − when one backend isn't enough...

This is a rather advanced watcher type that lets you embed one event loop into another (currently only IO events are supported in the embedded loop, other types of watchers might be handled in a delayed or incorrect fashion and must not be used).

See the libev documentation at <http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#code_ev_embed_code_when_one_backend_> (locally installed as *EV::libev*) for more details.

In short, this watcher is most useful on BSD systems without working kqueue to still be able to handle a large number of sockets:

```
my $socket_loop;

# check whether we use SELECT or POLL _and_ KQUEUE is supported
if (
  (EV::backend & (EV::BACKEND_POLL | EV::BACKEND_SELECT))
  && (EV::supported_backends & EV::embeddable_backends & EV::BACKEND_KQUEUE)
) {
  # use kqueue for sockets
  $socket_loop = new EV::Loop EV::BACKEND_KQUEUE | EV::FLAG_NOENV;
}

# use the default loop otherwise
$socket_loop ||= EV::default_loop;
```

`$w` = EV::embed `$otherloop`[, `$callback`]
`$w` = EV::embed_ns `$otherloop`[, `$callback`]
`$w` = `$loop`−>embed ($otherloop[, `$callback`])
`$w` = `$loop`−>embed_ns ($otherloop[, `$callback`])

Call the callback when the embedded event loop (`$otherloop`) has any I/O activity. The `$callback` is optional: if it is missing, then the embedded event loop will be managed automatically (which is recommended), otherwise you have to invoke `sweep` yourself.

The `embed_ns` variant doesn't start (activate) the newly created watcher.

ASYNC WATCHERS − how to wake up another event loop

Async watchers are provided by EV, but have little use in perl directly, as perl neither supports threads running in parallel nor direct access to signal handlers or other contexts where they could be of value.

It is, however, possible to use them from the XS level.

Please see the libev documentation for further details.

`$w` = EV::async `$callback`
`$w` = EV::async_ns `$callback`

```
$w = $loop->async ($callback)
$w = $loop->async_ns ($callback)
$w->send
$bool = $w->async_pending
```

CLEANUP WATCHERS – how to clean up when the event loop goes away

Cleanup watchers are not supported on the Perl level, they can only be used via XS currently.

PERL SIGNALS

While Perl signal handling (`%SIG`) is not affected by EV, the behaviour with EV is as the same as any other C library: Perl-signals will only be handled when Perl runs, which means your signal handler might be invoked only the next time an event callback is invoked.

The solution is to use EV signal watchers (see `EV::signal`), which will ensure proper operations with regards to other event watchers.

If you cannot do this for whatever reason, you can also force a watcher to be called on every event loop iteration by installing a `EV::check` watcher:

```
my $async_check = EV::check sub { };
```

This ensures that perl gets into control for a short time to handle any pending signals, and also ensures (slightly) slower overall operation.

ITHREADS

Ithreads are not supported by this module in any way. Perl pseudo-threads is evil stuff and must die. Real threads as provided by Coro are fully supported (and enhanced support is available via Coro::EV).

FORK

Most of the "improved" event delivering mechanisms of modern operating systems have quite a few problems with **fork**(2) (to put it bluntly: it is not supported and usually destructive). Libev makes it possible to work around this by having a function that recreates the kernel state after fork in the child.

On non-win32 platforms, this module requires the pthread_atfork functionality to do this automatically for you. This function is quite buggy on most BSDs, though, so YMMV. The overhead for this is quite negligible, because everything the function currently does is set a flag that is checked only when the event loop gets used the next time, so when you do fork but not use EV, the overhead is minimal.

On win32, there is no notion of fork so all this doesn't apply, of course.

SEE ALSO

EV::MakeMaker – MakeMaker interface to XS API, EV::ADNS (asynchronous DNS), Glib::EV (makes Glib/Gtk2 use EV as event loop), EV::Glib (embed Glib into EV), Coro::EV (efficient thread integration), Net::SNMP::EV (asynchronous SNMP), AnyEvent for event-loop agnostic and portable event driven programming.

AUTHOR

Marc Lehmann <schmorp@schmorp.de>
http://home.schmorp.de/