NAME

libev – a high performance full–featured event loop written in C

SYNOPSIS

```
#include <ev.h>
```

EXAMPLE PROGRAM

```
// a single header file is required
#include <ev.h>

#include <stdio.h> // for puts

// every watcher type has its own typedef'd struct
// with the name ev_TYPE
ev_io stdin_watcher;
ev_timer timeout_watcher;

// all watcher callbacks have a similar signature
// this callback is called when data is readable on stdin
static void
stdin_cb (EV_P_ ev_io *w, int revents)
{
  puts ("stdin ready");
  // for one-shot events, one must manually stop the watcher
  // with its corresponding stop function.
  ev_io_stop (EV_A_ w);

  // this causes all nested ev_run's to stop iterating
  ev_break (EV_A_ EVBREAK_ALL);
}

// another callback, this time for a time-out
static void
timeout_cb (EV_P_ ev_timer *w, int revents)
{
  puts ("timeout");
  // this causes the innermost ev_run to stop iterating
  ev_break (EV_A_ EVBREAK_ONE);
}

int
main (void)
{
  // use the default event loop unless you have special needs
  struct ev_loop *loop = EV_DEFAULT;

  // initialise an io watcher, then start it
  // this one will watch for stdin to become readable
  ev_io_init (&stdin_watcher, stdin_cb, /*STDIN_FILENO*/ 0, EV_READ);
  ev_io_start (loop, &stdin_watcher);

  // initialise a timer watcher, then start it
  // simple non-repeating 5.5 second timeout
  ev_timer_init (&timeout_watcher, timeout_cb, 5.5, 0.);
  ev_timer_start (loop, &timeout_watcher);
```

```
  // now wait for events to arrive
  ev_run (loop, 0);

  // break was called, so exit
  return 0;
}
```

ABOUT THIS DOCUMENT

This document documents the libev software package.

The newest version of this document is also available as an html-formatted web page you might find easier to navigate when reading it for the first time: <http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod>.

While this document tries to be as complete as possible in documenting libev, its usage and the rationale behind its design, it is not a tutorial on event-based programming, nor will it introduce event-based programming with libev.

Familiarity with event based programming techniques in general is assumed throughout this document.

WHAT TO READ WHEN IN A HURRY

This manual tries to be very detailed, but unfortunately, this also makes it very long. If you just want to know the basics of libev, I suggest reading "ANATOMY OF A WATCHER", then the "EXAMPLE PROGRAM" above and look up the missing functions in "GLOBAL FUNCTIONS" and the `ev_io` and `ev_timer` sections in "WATCHER TYPES".

ABOUT LIBEV

Libev is an event loop: you register interest in certain events (such as a file descriptor being readable or a timeout occurring), and it will manage these event sources and provide your program with events.

To do this, it must take more or less complete control over your process (or thread) by executing the *event loop* handler, and will then communicate events via a callback mechanism.

You register interest in certain events by registering so-called *event watchers*, which are relatively small C structures you initialise with the details of the event, and then hand it over to libev by *starting* the watcher.

FEATURES

Libev supports `select`, `poll`, the Linux-specific aio and `epoll` interfaces, the BSD-specific `kqueue` and the Solaris-specific event port mechanisms for file descriptor events (`ev_io`), the Linux `inotify` interface (for `ev_stat`), Linux eventfd/signalfd (for faster and cleaner inter-thread wakeup (`ev_async`)/signal handling (`ev_signal`)) relative timers (`ev_timer`), absolute timers with customised rescheduling (`ev_periodic`), synchronous signals (`ev_signal`), process status change events (`ev_child`), and event watchers dealing with the event loop mechanism itself (`ev_idle`, `ev_embed`, `ev_prepare` and `ev_check` watchers) as well as file watchers (`ev_stat`) and even limited support for fork events (`ev_fork`).

It also is quite fast (see this benchmark <http://libev.schmorp.de/bench.html> comparing it to libevent for example).

CONVENTIONS

Libev is very configurable. In this manual the default (and most common) configuration will be described, which supports multiple event loops. For more info about various configuration options please have a look at **EMBED** section in this manual. If libev was configured without support for multiple event loops, then all functions taking an initial argument of name `loop` (which is always of type `struct ev_loop *`) will not have this argument.

TIME REPRESENTATION

Libev represents time as a single floating point number, representing the (fractional) number of seconds since the (POSIX) epoch (in practice somewhere near the beginning of 1970, details are complicated, don't ask). This type is called `ev_tstamp`, which is what you should use too. It usually aliases to the `double` type in C. When you need to do any calculations on it, you should treat it as some floating point value.

Unlike the name component `stamp` might indicate, it is also used for time differences (e.g. delays)

throughout libev.

ERROR HANDLING

Libev knows three classes of errors: operating system errors, usage errors and internal errors (bugs).

When libev catches an operating system error it cannot handle (for example a system call indicating a condition libev cannot fix), it calls the callback set via `ev_set_syserr_cb`, which is supposed to fix the problem or abort. The default is to print a diagnostic message and to call `abort ()`.

When libev detects a usage error such as a negative timer interval, then it will print a diagnostic message and abort (via the `assert` mechanism, so `NDEBUG` will disable this checking): these are programming errors in the libev caller and need to be fixed there.

Via the `EV_FREQUENT` macro you can compile in and/or enable extensive consistency checking code inside libev that can be used to check for internal inconsistencies, suually caused by application bugs.

Libev also has a few internal error-checking `assert`ions. These do not trigger under normal circumstances, as they indicate either a bug in libev or worse.

GLOBAL FUNCTIONS

These functions can be called anytime, even before initialising the library in any way.

ev_tstamp ev_time ()

Returns the current time as libev would use it. Please note that the `ev_now` function is usually faster and also often returns the timestamp you actually want to know. Also interesting is the combination of `ev_now_update` and `ev_now`.

ev_sleep (ev_tstamp interval)

Sleep for the given interval: The current thread will be blocked until either it is interrupted or the given time interval has passed (approximately – it might return a bit earlier even if not interrupted). Returns immediately if `interval <= 0`.

Basically this is a sub-second-resolution `sleep ()`.

The range of the `interval` is limited – libev only guarantees to work with sleep times of up to one day (`interval <= 86400`).

int ev_version_major ()

int ev_version_minor ()

You can find out the major and minor ABI version numbers of the library you linked against by calling the functions `ev_version_major` and `ev_version_minor`. If you want, you can compare against the global symbols `EV_VERSION_MAJOR` and `EV_VERSION_MINOR`, which specify the version of the library your program was compiled against.

These version numbers refer to the ABI version of the library, not the release version.

Usually, it's a good idea to terminate if the major versions mismatch, as this indicates an incompatible change. Minor versions are usually compatible to older versions, so a larger minor version alone is usually not a problem.

Example: Make sure we haven't accidentally been linked against the wrong version (note, however, that this will not detect other ABI mismatches, such as LFS or reentrancy).

```
assert (("libev version mismatch",
         ev_version_major () == EV_VERSION_MAJOR
         && ev_version_minor () >= EV_VERSION_MINOR));
```

unsigned int ev_supported_backends ()

Return the set of all backends (i.e. their corresponding `EV_BACKEND_*` value) compiled into this binary of libev (independent of their availability on the system you are running on). See `ev_default_loop` for a description of the set values.

Example: make sure we have the epoll method, because yeah this is cool and a must have and can we have a torrent of it please!!!11

```
assert (("sorry, no epoll, no sex",
         ev_supported_backends () & EVBACKEND_EPOLL));
```

unsigned int ev_recommended_backends ()

Return the set of all backends compiled into this binary of libev and also recommended for this platform, meaning it will work for most file descriptor types. This set is often smaller than the one returned by `ev_supported_backends`, as for example kqueue is broken on most BSDs and will not be auto-detected unless you explicitly request it (assuming you know what you are doing). This is the set of backends that libev will probe for if you specify no backends explicitly.

unsigned int ev_embeddable_backends ()

Returns the set of backends that are embeddable in other event loops. This value is platform-specific but can include backends not available on the current system. To find which embeddable backends might be supported on the current system, you would need to look at `ev_embeddable_backends () & ev_supported_backends ()`, likewise for recommended ones.

See the description of `ev_embed` watchers for more info.

ev_set_allocator (void *(*cb)(void *ptr, long size) throw ())

Sets the allocation function to use (the prototype is similar − the semantics are identical to the `realloc` C89/SuS/POSIX function). It is used to allocate and free memory (no surprises here). If it returns zero when memory needs to be allocated (`size != 0`), the library might abort or take some potentially destructive action.

Since some systems (at least OpenBSD and Darwin) fail to implement correct `realloc` semantics, libev will use a wrapper around the system `realloc` and `free` functions by default.

You could override this function in high-availability programs to, say, free some memory if it cannot allocate memory, to use a special allocator, or even to sleep a while and retry until some memory is available.

Example: The following is the `realloc` function that libev itself uses which should work with `realloc` and `free` functions of all kinds and is probably a good basis for your own implementation.

```
static void *
ev_realloc_emul (void *ptr, long size) EV_NOEXCEPT
{
  if (size)
    return realloc (ptr, size);

  free (ptr);
  return 0;
}
```

Example: Replace the libev allocator with one that waits a bit and then retries.

```
static void *
persistent_realloc (void *ptr, size_t size)
{
  if (!size)
    {
      free (ptr);
      return 0;
    }

  for (;;)
    {
      void *newptr = realloc (ptr, size);

      if (newptr)
```

```
            return newptr;

          sleep (60);
        }
    }

    ...
    ev_set_allocator (persistent_realloc);
```

ev_set_syserr_cb (void (*cb)(const char *msg) throw ())

Set the callback function to call on a retryable system call error (such as failed select, poll, epoll_wait). The message is a printable string indicating the system call or subsystem causing the problem. If this callback is set, then libev will expect it to remedy the situation, no matter what, when it returns. That is, libev will generally retry the requested operation, or, if the condition doesn't go away, do bad stuff (such as abort).

Example: This is basically the same thing that libev does internally, too.

```
static void
fatal_error (const char *msg)
{
  perror (msg);
  abort ();
}

...
ev_set_syserr_cb (fatal_error);
```

ev_feed_signal (int signum)

This function can be used to ''simulate'' a signal receive. It is completely safe to call this function at any time, from any context, including signal handlers or random threads.

Its main use is to customise signal handling in your process, especially in the presence of threads. For example, you could block signals by default in all threads (and specifying `EVFLAG_NOSIGMASK` when creating any loops), and in one thread, use `sigwait` or any other mechanism to wait for signals, then ''deliver'' them to libev by calling `ev_feed_signal`.

FUNCTIONS CONTROLLING EVENT LOOPS

An event loop is described by a `struct ev_loop *` (the `struct` is *not* optional in this case unless libev 3 compatibility is disabled, as libev 3 had an `ev_loop` function colliding with the struct name).

The library knows two types of such loops, the *default* loop, which supports child process events, and dynamically created event loops which do not.

struct ev_loop *ev_default_loop (unsigned int flags)

This returns the ''default'' event loop object, which is what you should normally use when you just need ''the event loop''. Event loop objects and the `flags` parameter are described in more detail in the entry for `ev_loop_new`.

If the default loop is already initialised then this function simply returns it (and ignores the flags. If that is troubling you, check `ev_backend ()` afterwards). Otherwise it will create it with the given flags, which should almost always be `0`, unless the caller is also the one calling `ev_run` or otherwise qualifies as ''the main program''.

If you don't know what event loop to use, use the one returned from this function (or via the `EV_DEFAULT` macro).

Note that this function is *not* thread-safe, so if you want to use it from multiple threads, you have to employ some kind of mutex (note also that this case is unlikely, as loops cannot be shared easily between threads anyway).

The default loop is the only loop that can handle `ev_child` watchers, and to do this, it always registers a handler for `SIGCHLD`. If this is a problem for your application you can either create a dynamic loop with `ev_loop_new` which doesn't do that, or you can simply overwrite the `SIGCHLD` signal handler *after* calling `ev_default_init`.

Example: This is the most typical usage.

```
if (!ev_default_loop (0))
  fatal ("could not initialise libev, bad $LIBEV_FLAGS in environment?");
```

Example: Restrict libev to the select and poll backends, and do not allow environment settings to be taken into account:

```
ev_default_loop (EVBACKEND_POLL | EVBACKEND_SELECT | EVFLAG_NOENV);
```

struct ev_loop *ev_loop_new (unsigned int flags)

This will create and initialise a new event loop object. If the loop could not be initialised, returns false.

This function is thread-safe, and one common way to use libev with threads is indeed to create one loop per thread, and using the default loop in the ''main'' or ''initial'' thread.

The flags argument can be used to specify special behaviour or specific backends to use, and is usually specified as `0` (or `EVFLAG_AUTO`).

The following flags are supported:

`EVFLAG_AUTO`

The default flags value. Use this if you have no clue (it's the right thing, believe me).

`EVFLAG_NOENV`

If this flag bit is or'ed into the flag value (or the program runs setuid or setgid) then libev will *not* look at the environment variable `LIBEV_FLAGS`. Otherwise (the default), this environment variable will override the flags completely if it is found in the environment. This is useful to try out specific backends to test their performance, to work around bugs, or to make libev threadsafe (accessing environment variables cannot be done in a threadsafe way, but usually it works if no other thread modifies them).

`EVFLAG_FORKCHECK`

Instead of calling `ev_loop_fork` manually after a fork, you can also make libev check for a fork in each iteration by enabling this flag.

This works by calling `getpid ()` on every iteration of the loop, and thus this might slow down your event loop if you do a lot of loop iterations and little real work, but is usually not noticeable (on my GNU/Linux system for example, `getpid` is actually a simple 5−insn sequence without a system call and thus *very* fast, but my GNU/Linux system also has `pthread_atfork` which is even faster). (Update: glibc versions 2.25 apparently removed the `getpid` optimisation again).

The big advantage of this flag is that you can forget about fork (and forget about forgetting to tell libev about forking, although you still have to ignore `SIGPIPE`) when you use this flag.

This flag setting cannot be overridden or specified in the `LIBEV_FLAGS` environment variable.

`EVFLAG_NOINOTIFY`

When this flag is specified, then libev will not attempt to use the *inotify* API for its `ev_stat` watchers. Apart from debugging and testing, this flag can be useful to conserve inotify file descriptors, as otherwise each loop using `ev_stat` watchers consumes one inotify handle.

`EVFLAG_SIGNALFD`

When this flag is specified, then libev will attempt to use the *signalfd* API for its `ev_signal` (and `ev_child`) watchers. This API delivers signals synchronously, which makes it both faster and might make it possible to get the queued signal data. It can also simplify signal handling with threads, as long as you properly block signals in your threads that are not interested in handling them.

Signalfd will not be used by default as this changes your signal mask, and there are a lot of shoddy libraries and programs (glib's threadpool for example) that can't properly initialise their signal masks.

`EVFLAG_NOSIGMASK`
When this flag is specified, then libev will avoid to modify the signal mask. Specifically, this means you have to make sure signals are unblocked when you want to receive them.

This behaviour is useful when you want to do your own signal handling, or want to handle signals only in specific threads and want to avoid libev unblocking the signals.

It's also required by POSIX in a threaded program, as libev calls `sigprocmask`, whose behaviour is officially unspecified.

This flag's behaviour will become the default in future versions of libev.

`EVBACKEND_SELECT` (value 1, portable select backend)
This is your standard **select** (2) backend. Not *completely* standard, as libev tries to roll its own fd_set with no limits on the number of fds, but if that fails, expect a fairly low limit on the number of fds when using this backend. It doesn't scale too well (O(highest_fd)), but its usually the fastest backend for a low number of (low-numbered :) fds.

To get good performance out of this backend you need a high amount of parallelism (most of the file descriptors should be busy). If you are writing a server, you should `accept ()` in a loop to accept as many connections as possible during one iteration. You might also want to have a look at `ev_set_io_collect_interval ()` to increase the amount of readiness notifications you get per iteration.

This backend maps `EV_READ` to the `readfds` set and `EV_WRITE` to the `writefds` set (and to work around Microsoft Windows bugs, also onto the `exceptfds` set on that platform).

`EVBACKEND_POLL` (value 2, poll backend, available everywhere except on windows)
And this is your standard **poll** (2) backend. It's more complicated than select, but handles sparse fds better and has no artificial limit on the number of fds you can use (except it will slow down considerably with a lot of inactive fds). It scales similarly to select, i.e. O(total_fds). See the entry for `EVBACKEND_SELECT`, above, for performance tips.

This backend maps `EV_READ` to `POLLIN | POLLERR | POLLHUP`, and `EV_WRITE` to `POLLOUT | POLLERR | POLLHUP`.

`EVBACKEND_EPOLL` (value 4, Linux)
Use the Linux-specific **epoll** (7) interface (for both pre− and post−2.6.9 kernels).

For few fds, this backend is a bit little slower than poll and select, but it scales phenomenally better. While poll and select usually scale like O(total_fds) where total_fds is the total number of fds (or the highest fd), epoll scales either O(1) or O(active_fds).

The epoll mechanism deserves honorable mention as the most misdesigned of the more advanced event mechanisms: mere annoyances include silently dropping file descriptors, requiring a system call per change per file descriptor (and unnecessary guessing of parameters), problems with dup, returning before the timeout value, resulting in additional iterations (and only giving 5ms accuracy while select on the same platform gives 0.1ms) and so on. The biggest issue is fork races, however − if a program forks then *both* parent and child process have to recreate the epoll set, which can take considerable time (one syscall per file descriptor) and is of course hard to detect.

Epoll is also notoriously buggy − embedding epoll fds *should* work, but of course *doesn't*, and epoll just loves to report events for totally *different* file descriptors (even already closed ones, so one cannot even remove them from the set) than registered in the set (especially on SMP systems). Libev tries to counter these spurious notifications by employing an additional generation counter and comparing that against the events to filter out spurious ones, recreating the set when required.

Epoll also erroneously rounds down timeouts, but gives you no way to know when and by how much, so sometimes you have to busy-wait because epoll returns immediately despite a nonzero timeout. And last not least, it also refuses to work with some file descriptors which work perfectly fine with `select` (files, many character devices...).

Epoll is truly the train wreck among event poll mechanisms, a frankenpoll, cobbled together in a hurry, no thought to design or interaction with others. Oh, the pain, will it ever stop...

While stopping, setting and starting an I/O watcher in the same iteration will result in some caching, there is still a system call per such incident (because the same *file descriptor* could point to a different *file description* now), so its best to avoid that. Also, `dup ()`'ed file descriptors might not work very well if you register events for both file descriptors.

Best performance from this backend is achieved by not unregistering all watchers for a file descriptor until it has been closed, if possible, i.e. keep at least one watcher active per fd at all times. Stopping and starting a watcher (without re-setting it) also usually doesn't cause extra overhead. A fork can both result in spurious notifications as well as in libev having to destroy and recreate the epoll object, which can take considerable time and thus should be avoided.

All this means that, in practice, `EVBACKEND_SELECT` can be as fast or faster than epoll for maybe up to a hundred file descriptors, depending on the usage. So sad.

While nominally embeddable in other event loops, this feature is broken in a lot of kernel revisions, but probably(!) works in current versions.

This backend maps `EV_READ` and `EV_WRITE` in the same way as `EVBACKEND_POLL`.

`EVBACKEND_LINUXAIO` (value 64, Linux)

Use the Linux-specific Linux AIO (*not* `aio(7)` but `io_submit(2)`) event interface available in post−4.18 kernels (but libev only tries to use it in 4.19+).

This is another Linux train wreck of an event interface.

If this backend works for you (as of this writing, it was very experimental), it is the best event interface available on Linux and might be well worth enabling it − if it isn't available in your kernel this will be detected and this backend will be skipped.

This backend can batch oneshot requests and supports a user-space ring buffer to receive events. It also doesn't suffer from most of the design problems of epoll (such as not being able to remove event sources from the epoll set), and generally sounds too good to be true. Because, this being the Linux kernel, of course it suffers from a whole new set of limitations, forcing you to fall back to epoll, inheriting all its design issues.

For one, it is not easily embeddable (but probably could be done using an event fd at some extra overhead). It also is subject to a system wide limit that can be configured in */proc/sys/fs/aio−max−nr*. If no AIO requests are left, this backend will be skipped during initialisation, and will switch to epoll when the loop is active.

Most problematic in practice, however, is that not all file descriptors work with it. For example, in Linux 5.1, TCP sockets, pipes, event fds, files, */dev/null* and many others are supported, but ttys do not work properly (a known bug that the kernel developers don't care about, see <https://lore.kernel.org/patchwork/patch/1047453/>), so this is not (yet?) a generic event polling interface.

Overall, it seems the Linux developers just don't want it to have a generic event handling mechanism other than `select` or `poll`.

To work around all these problem, the current version of libev uses its epoll backend as a fallback for file descriptor types that do not work. Or falls back completely to epoll if the kernel acts up.

This backend maps `EV_READ` and `EV_WRITE` in the same way as `EVBACKEND_POLL`.

EVBACKEND_KQUEUE (value 8, most BSD clones)
Kqueue deserves special mention, as at the time this backend was implemented, it was broken on all BSDs except NetBSD (usually it doesn't work reliably with anything but sockets and pipes, except on Darwin, where of course it's completely useless). Unlike epoll, however, whose brokenness is by design, these kqueue bugs can be (and mostly have been) fixed without API changes to existing programs. For this reason it's not being "auto-detected" on all platforms unless you explicitly specify it in the flags (i.e. using EVBACKEND_KQUEUE) or libev was compiled on a known-to-be-good (-enough) system like NetBSD.

You still can embed kqueue into a normal poll or select backend and use it only for sockets (after having made sure that sockets work with kqueue on the target platform). See ev_embed watchers for more info.

It scales in the same way as the epoll backend, but the interface to the kernel is more efficient (which says nothing about its actual speed, of course). While stopping, setting and starting an I/O watcher does never cause an extra system call as with EVBACKEND_EPOLL, it still adds up to two event changes per incident. Support for fork () is very bad (you might have to leak fds on fork, but it's more sane than epoll) and it drops fds silently in similarly hard-to-detect cases.

This backend usually performs well under most conditions.

While nominally embeddable in other event loops, this doesn't work everywhere, so you might need to test for this. And since it is broken almost everywhere, you should only use it when you have a lot of sockets (for which it usually works), by embedding it into another event loop (e.g. EVBACKEND_SELECT or EVBACKEND_POLL (but poll is of course also broken on OS X)) and, did I mention it, using it only for sockets.

This backend maps EV_READ into an EVFILT_READ kevent with NOTE_EOF, and EV_WRITE into an EVFILT_WRITE kevent with NOTE_EOF.

EVBACKEND_DEVPOLL (value 16, Solaris 8)
This is not implemented yet (and might never be, unless you send me an implementation). According to reports, /dev/poll only supports sockets and is not embeddable, which would limit the usefulness of this backend immensely.

EVBACKEND_PORT (value 32, Solaris 10)
This uses the Solaris 10 event port mechanism. As with everything on Solaris, it's really slow, but it still scales very well (O(active_fds)).

While this backend scales well, it requires one system call per active file descriptor per loop iteration. For small and medium numbers of file descriptors a "slow" EVBACKEND_SELECT or EVBACKEND_POLL backend might perform better.

On the positive side, this backend actually performed fully to specification in all tests and is fully embeddable, which is a rare feat among the OS-specific backends (I vastly prefer correctness over speed hacks).

On the negative side, the interface is *bizarre* - so bizarre that even sun itself gets it wrong in their code examples: The event polling function sometimes returns events to the caller even though an error occurred, but with no indication whether it has done so or not (yes, it's even documented that way) - deadly for edge-triggered interfaces where you absolutely have to know whether an event occurred or not because you have to re-arm the watcher.

Fortunately libev seems to be able to work around these idiocies.

This backend maps EV_READ and EV_WRITE in the same way as EVBACKEND_POLL.

EVBACKEND_ALL
Try all backends (even potentially broken ones that wouldn't be tried with EVFLAG_AUTO). Since this is a mask, you can do stuff such as EVBACKEND_ALL & ~EVBACKEND_KQUEUE.

It is definitely not recommended to use this flag, use whatever ev_recommended_backends

() returns, or simply do not specify a backend at all.

EVBACKEND_MASK
Not a backend at all, but a mask to select all backend bits from a flags value, in case you want to mask out any backends from a flags value (e.g. when modifying the LIBEV_FLAGS environment variable).

If one or more of the backend flags are or'ed into the flags value, then only these backends will be tried (in the reverse order as listed here). If none are specified, all backends in ev_recommended_backends () will be tried.

Example: Try to create a event loop that uses epoll and nothing else.

```
struct ev_loop *epoller = ev_loop_new (EVBACKEND_EPOLL | EVFLAG_NOENV);
if (!epoller)
  fatal ("no epoll found here, maybe it hides under your chair");
```

Example: Use whatever libev has to offer, but make sure that kqueue is used if available.

```
struct ev_loop *loop = ev_loop_new (ev_recommended_backends () | EVBACKEND_
```

Example: Similarly, on linux, you mgiht want to take advantage of the linux aio backend if possible, but fall back to something else if that isn't available.

```
struct ev_loop *loop = ev_loop_new (ev_recommended_backends () | EVBACKEND_
```

ev_loop_destroy (loop)
Destroys an event loop object (frees all memory and kernel state etc.). None of the active event watchers will be stopped in the normal sense, so e.g. ev_is_active might still return true. It is your responsibility to either stop all watchers cleanly yourself *before* calling this function, or cope with the fact afterwards (which is usually the easiest thing, you can just ignore the watchers and/or free () them for example).

Note that certain global state, such as signal state (and installed signal handlers), will not be freed by this function, and related watchers (such as signal and child watchers) would need to be stopped manually.

This function is normally used on loop objects allocated by ev_loop_new, but it can also be used on the default loop returned by ev_default_loop, in which case it is not thread-safe.

Note that it is not advisable to call this function on the default loop except in the rare occasion where you really need to free its resources. If you need dynamically allocated loops it is better to use ev_loop_new and ev_loop_destroy.

ev_loop_fork (loop)
This function sets a flag that causes subsequent ev_run iterations to reinitialise the kernel state for backends that have one. Despite the name, you can call it anytime you are allowed to start or stop watchers (except inside an ev_prepare callback), but it makes most sense after forking, in the child process. You *must* call it (or use EVFLAG_FORKCHECK) in the child before resuming or calling ev_run.

In addition, if you want to reuse a loop (via this function or EVFLAG_FORKCHECK), you *also* have to ignore SIGPIPE.

Again, you *have* to call it on *any* loop that you want to re-use after a fork, *even if you do not plan to use the loop in the parent*. This is because some kernel interfaces *cough* *kqueue* *cough* do funny things during fork.

On the other hand, you only need to call this function in the child process if and only if you want to use the event loop in the child. If you just fork+exec or create a new loop in the child, you don't have to call it at all (in fact, epoll is so badly broken that it makes a difference, but libev will usually detect this case on its own and do a costly reset of the backend).

The function itself is quite fast and it's usually not a problem to call it just in case after a fork.

Example: Automate calling `ev_loop_fork` on the default loop when using pthreads.

```
static void
post_fork_child (void)
{
  ev_loop_fork (EV_DEFAULT);
}

...
pthread_atfork (0, 0, post_fork_child);
```

int ev_is_default_loop (loop)
: Returns true when the given loop is, in fact, the default loop, and false otherwise.

unsigned int ev_iteration (loop)
: Returns the current iteration count for the event loop, which is identical to the number of times libev did poll for new events. It starts at `0` and happily wraps around with enough iterations.

  This value can sometimes be useful as a generation counter of sorts (it "ticks" the number of loop iterations), as it roughly corresponds with `ev_prepare` and `ev_check` calls - and is incremented between the prepare and check phases.

unsigned int ev_depth (loop)
: Returns the number of times `ev_run` was entered minus the number of times `ev_run` was exited normally, in other words, the recursion depth.

  Outside `ev_run`, this number is zero. In a callback, this number is `1`, unless `ev_run` was invoked recursively (or from another thread), in which case it is higher.

  Leaving `ev_run` abnormally (setjmp/longjmp, cancelling the thread, throwing an exception etc.), doesn't count as "exit" - consider this as a hint to avoid such ungentleman-like behaviour unless it's really convenient, in which case it is fully supported.

unsigned int ev_backend (loop)
: Returns one of the `EVBACKEND_*` flags indicating the event backend in use.

ev_tstamp ev_now (loop)
: Returns the current "event loop time", which is the time the event loop received events and started processing them. This timestamp does not change as long as callbacks are being processed, and this is also the base time used for relative timers. You can treat it as the timestamp of the event occurring (or more correctly, libev finding out about it).

ev_now_update (loop)
: 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.

  See also "The special problem of time updates" in the `ev_timer` section.

ev_suspend (loop)
ev_resume (loop)
: These two functions suspend and resume an event 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 `ev_suspend` in your `SIGTSTP` handler, sending yourself a `SIGSTOP` and calling `ev_resume` directly afterwards to resume timer processing.

Effectively, all `ev_timer` watchers will be delayed by the time spend between `ev_suspend` and `ev_resume`, and all `ev_periodic` watchers will be rescheduled (that is, they will lose any events that would have occurred while suspended).

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

Calling `ev_suspend`/`ev_resume` has the side effect of updating the event loop time (see `ev_now_update`).

bool ev_run (loop, int flags)

Finally, this is it, the event handler. This function usually is called after you have initialised all your watchers and you want to start handling events. It will ask the operating system for any new events, call the watcher callbacks, and then repeat the whole process indefinitely: This is why event loops are called *loops*.

If the flags argument is specified as `0`, it will keep handling events until either no event watchers are active anymore or `ev_break` was called.

The return value is false if there are no more active watchers (which usually means ‘‘all jobs done’’ or ‘‘deadlock’’), and true in all other cases (which usually means " you should call `ev_run` again").

Please note that an explicit `ev_break` is usually better than relying on all watchers to be stopped when deciding when a program has finished (especially in interactive programs), but having a program that automatically loops as long as it has to and no longer by virtue of relying on its watchers stopping correctly, that is truly a thing of beauty.

This function is *mostly* exception-safe – you can break out of a `ev_run` call by calling `longjmp` in a callback, throwing a C++ exception and so on. This does not decrement the `ev_depth` value, nor will it clear any outstanding `EVBREAK_ONE` breaks.

A flags value of `EVRUN_NOWAIT` will look for new events, will handle those events and any already outstanding ones, but will not wait and block your process in case there are no events and will return after one iteration of the loop. This is sometimes useful to poll and handle new events while doing lengthy calculations, to keep the program responsive.

A flags value of `EVRUN_ONCE` will look for new events (waiting if necessary) and will handle those and any already outstanding ones. It will block your process until at least one new event arrives (which could be an event internal to libev itself, so there is no guarantee that a user-registered callback will be called), and will return after one iteration of the loop.

This is useful if you are waiting for some external event in conjunction with something not expressible using other libev watchers (i.e. "roll your own `ev_run`"). However, a pair of `ev_prepare`/`ev_check` watchers is usually a better approach for this kind of thing.

Here are the gory details of what `ev_run` does (this is for your understanding, not a guarantee that things will work exactly like this in future versions):

```
- Increment loop depth.
- Reset the ev_break status.
- Before the first iteration, call any pending watchers.
LOOP:
- If EVFLAG_FORKCHECK was used, check for a fork.
- If a fork was detected (by any means), queue and call all fork watchers.
- Queue and call all prepare watchers.
- If ev_break was called, goto FINISH.
- If we have been forked, detach and recreate the kernel state
  as to not disturb the other process.
- Update the kernel state with all outstanding changes.
- Update the "event loop time" (ev_now ()).
- Calculate for how long to sleep or block, if at all
```

```
   (active idle watchers, EVRUN_NOWAIT or not having
   any active watchers at all will result in not sleeping).
- Sleep if the I/O and timer collect interval say so.
- Increment loop iteration counter.
- Block the process, waiting for any events.
- Queue all outstanding I/O (fd) events.
- Update the "event loop time" (ev_now ()), and do time jump adjustments.
- Queue all expired timers.
- Queue all expired periodics.
- Queue all idle watchers with priority higher than that of pending events.
- Queue all check watchers.
- Call all queued watchers in reverse order (i.e. check watchers first).
   Signals and child watchers are implemented as I/O watchers, and will
   be handled here by queueing them when their watcher gets executed.
- If ev_break has been called, or EVRUN_ONCE or EVRUN_NOWAIT
   were used, or there are no active watchers, goto FINISH, otherwise
   continue with step LOOP.
FINISH:
- Reset the ev_break status iff it was EVBREAK_ONE.
- Decrement the loop depth.
- Return.
```

Example: Queue some jobs and then loop until no events are outstanding anymore.

```
... queue jobs here, make sure they register event watchers as long
... as they still have work to do (even an idle watcher will do..)
ev_run (my_loop, 0);
... jobs done or somebody called break. yeah!
```

ev_break (loop, how)

Can be used to make a call to `ev_run` return early (but only after it has processed all outstanding events). The `how` argument must be either `EVBREAK_ONE`, which will make the innermost `ev_run` call return, or `EVBREAK_ALL`, which will make all nested `ev_run` calls return.

This "break state" will be cleared on the next call to `ev_run`.

It is safe to call `ev_break` from outside any `ev_run` calls, too, in which case it will have no effect.

ev_ref (loop)

ev_unref (loop)

Ref/unref can be used to add or remove a reference count on the event loop: Every watcher keeps one reference, and as long as the reference count is nonzero, `ev_run` will not return on its own.

This is useful when you have a watcher that you never intend to unregister, but that nevertheless should not keep `ev_run` from returning. In such a case, call `ev_unref` after starting, and `ev_ref` before stopping it.

As an example, libev itself uses this for its internal signal pipe: It is not visible to the libev user and should not keep `ev_run` from exiting if no event watchers registered by it are active. It is also an excellent way to do this for generic recurring timers or from within third-party libraries. Just remember to *unref after start* and *ref before stop* (but only if the watcher wasn't active before, or was active before, respectively. Note also that libev might stop watchers itself (e.g. non-repeating timers) in which case you have to `ev_ref` in the callback).

Example: Create a signal watcher, but keep it from keeping `ev_run` running when nothing else is active.

```
ev_signal exitsig;
ev_signal_init (&exitsig, sig_cb, SIGINT);
ev_signal_start (loop, &exitsig);
ev_unref (loop);
```

Example: For some weird reason, unregister the above signal handler again.

```
ev_ref (loop);
ev_signal_stop (loop, &exitsig);
```

ev_set_io_collect_interval (loop, ev_tstamp interval)

ev_set_timeout_collect_interval (loop, ev_tstamp interval)

These advanced functions influence the time that libev will spend waiting for events. Both time intervals are by default `0`, meaning that libev will try to invoke timer/periodic callbacks and I/O callbacks with minimum latency.

Setting these to a higher value (the `interval` *must* be >= `0`) allows libev to delay invocation of I/O and timer/periodic callbacks to increase efficiency of loop iterations (or to increase power-saving opportunities).

The idea is that sometimes your program runs just fast enough to handle one (or very few) event(s) per loop iteration. While this makes the program responsive, it also wastes a lot of CPU time to poll for new events, especially with backends like `select ()` which have a high overhead for the actual polling but can deliver many events at once.

By setting a higher *io collect interval* you allow libev to spend more time collecting I/O events, so you can handle more events per iteration, at the cost of increasing latency. Timeouts (both `ev_periodic` and `ev_timer`) will not be affected. Setting this to a non-null value will introduce an additional `ev_sleep ()` call into most loop iterations. The sleep time ensures that libev will not poll for I/O events more often then once per this interval, on average (as long as the host time resolution is good enough).

Likewise, by setting a higher *timeout collect interval* you allow libev to spend more time collecting timeouts, at the expense of increased latency/jitter/inexactness (the watcher callback will be called later). `ev_io` watchers will not be affected. Setting this to a non-null value will not introduce any overhead in libev.

Many (busy) programs can usually benefit by setting the I/O collect interval to a value near `0.1` or so, which is often enough for interactive servers (of course not for games), likewise for timeouts. It usually doesn't make much sense to set it to a lower value than `0.01`, as this approaches the timing granularity of most systems. Note that if you do transactions with the outside world and you can't increase the parallelity, then this setting will limit your transaction rate (if you need to poll once per transaction and the I/O collect interval is 0.01, then you can't do more than 100 transactions per second).

Setting the *timeout collect interval* can improve the opportunity for saving power, as the program will "bundle" timer callback invocations that are "near" in time together, by delaying some, thus reducing the number of times the process sleeps and wakes up again. Another useful technique to reduce iterations/wake-ups is to use `ev_periodic` watchers and make sure they fire on, say, one-second boundaries only.

Example: we only need 0.1s timeout granularity, and we wish not to poll more often than 100 times per second:

```
ev_set_timeout_collect_interval (EV_DEFAULT_UC_ 0.1);
ev_set_io_collect_interval (EV_DEFAULT_UC_ 0.01);
```

ev_invoke_pending (loop)

This call will simply invoke all pending watchers while resetting their pending state. Normally, `ev_run` does this automatically when required, but when overriding the invoke callback this call comes handy. This function can be invoked from a watcher - this can be useful for example when you

want to do some lengthy calculation and want to pass further event handling to another thread (you still have to make sure only one thread executes within `ev_invoke_pending` or `ev_run` of course).

int ev_pending_count (loop)
Returns the number of pending watchers – zero indicates that no watchers are pending.

ev_set_invoke_pending_cb (loop, void (*invoke_pending_cb)(EV_P))
This overrides the invoke pending functionality of the loop: Instead of invoking all pending watchers when there are any, `ev_run` will call this callback instead. This is useful, for example, when you want to invoke the actual watchers inside another context (another thread etc.).

If you want to reset the callback, use `ev_invoke_pending` as new callback.

ev_set_loop_release_cb (loop, void (*release)(EV_P) throw (), void (*acquire)(EV_P) throw ())
Sometimes you want to share the same loop between multiple threads. This can be done relatively simply by putting mutex_lock/unlock calls around each call to a libev function.

However, `ev_run` can run an indefinite time, so it is not feasible to wait for it to return. One way around this is to wake up the event loop via `ev_break` and `ev_async_send`, another way is to set these *release* and *acquire* callbacks on the loop.

When set, then `release` will be called just before the thread is suspended waiting for new events, and `acquire` is called just afterwards.

Ideally, `release` will just call your mutex_unlock function, and `acquire` will just call the mutex_lock function again.

While event loop modifications are allowed between invocations of `release` and `acquire` (that's their only purpose after all), no modifications done will affect the event loop, i.e. adding watchers will have no effect on the set of file descriptors being watched, or the time waited. Use an `ev_async` watcher to wake up `ev_run` when you want it to take note of any changes you made.

In theory, threads executing `ev_run` will be async-cancel safe between invocations of `release` and `acquire`.

See also the locking example in the `THREADS` section later in this document.

ev_set_userdata (loop, void *data)
void *ev_userdata (loop)
Set and retrieve a single `void *` associated with a loop. When `ev_set_userdata` has never been called, then `ev_userdata` returns `0`.

These two functions can be used to associate arbitrary data with a loop, and are intended solely for the `invoke_pending_cb`, `release` and `acquire` callbacks described above, but of course can be (ab-)used for any other purpose as well.

ev_verify (loop)
This function only does something when `EV_VERIFY` support has been compiled in, which is the default for non-minimal builds. It tries to go through all internal structures and checks them for validity. If anything is found to be inconsistent, it will print an error message to standard error and call `abort ()`.

This can be used to catch bugs inside libev itself: under normal circumstances, this function will never abort as of course libev keeps its data structures consistent.

ANATOMY OF A WATCHER

In the following description, uppercase `TYPE` in names stands for the watcher type, e.g. `ev_TYPE_start` can mean `ev_timer_start` for timer watchers and `ev_io_start` for I/O watchers.

A watcher is an opaque structure that you allocate and register to record your interest in some event. To make a concrete example, imagine you want to wait for STDIN to become readable, you would create an `ev_io` watcher for that:

```
static void my_cb (struct ev_loop *loop, ev_io *w, int revents)
{
  ev_io_stop (w);
  ev_break (loop, EVBREAK_ALL);
}

struct ev_loop *loop = ev_default_loop (0);

ev_io stdin_watcher;

ev_init (&stdin_watcher, my_cb);
ev_io_set (&stdin_watcher, STDIN_FILENO, EV_READ);
ev_io_start (loop, &stdin_watcher);

ev_run (loop, 0);
```

As you can see, you are responsible for allocating the memory for your watcher structures (and it is *usually* a bad idea to do this on the stack).

Each watcher has an associated watcher structure (called `struct ev_TYPE` or simply `ev_TYPE`, as typedefs are provided for all watcher structs).

Each watcher structure must be initialised by a call to `ev_init (watcher *, callback)`, which expects a callback to be provided. This callback is invoked each time the event occurs (or, in the case of I/O watchers, each time the event loop detects that the file descriptor given is readable and/or writable).

Each watcher type further has its own `ev_TYPE_set (watcher *, ...)` macro to configure it, with arguments specific to the watcher type. There is also a macro to combine initialisation and setting in one call: `ev_TYPE_init (watcher *, callback, ...)`.

To make the watcher actually watch out for events, you have to start it with a watcher-specific start function (`ev_TYPE_start (loop, watcher *)`), and you can stop watching for events at any time by calling the corresponding stop function (`ev_TYPE_stop (loop, watcher *)`.

As long as your watcher is active (has been started but not stopped) you must not touch the values stored in it. Most specifically you must never reinitialise it or call its `ev_TYPE_set` macro.

Each and every callback receives the event loop pointer as first, the registered watcher structure as second, and a bitset of received events as third argument.

The received events usually include a single bit per event type received (you can receive multiple events at the same time). The possible bit masks are:

`EV_READ`
`EV_WRITE`
: The file descriptor in the `ev_io` watcher has become readable and/or writable.

`EV_TIMER`
: The `ev_timer` watcher has timed out.

`EV_PERIODIC`
: The `ev_periodic` watcher has timed out.

`EV_SIGNAL`
: The signal specified in the `ev_signal` watcher has been received by a thread.

`EV_CHILD`
: The pid specified in the `ev_child` watcher has received a status change.

`EV_STAT`
: The path specified in the `ev_stat` watcher changed its attributes somehow.

`EV_IDLE`
The `ev_idle` watcher has determined that you have nothing better to do.

`EV_PREPARE`
`EV_CHECK`
All `ev_prepare` watchers are invoked just *before* `ev_run` starts to gather new events, and all `ev_check` watchers are queued (not invoked) just after `ev_run` has gathered them, but before it queues any callbacks for any received events. That means `ev_prepare` watchers are the last watchers invoked before the event loop sleeps or polls for new events, and `ev_check` watchers will be invoked before any other watchers of the same or lower priority within an event loop iteration.

Callbacks of both watcher types can start and stop as many watchers as they want, and all of them will be taken into account (for example, a `ev_prepare` watcher might start an idle watcher to keep `ev_run` from blocking).

`EV_EMBED`
The embedded event loop specified in the `ev_embed` watcher needs attention.

`EV_FORK`
The event loop has been resumed in the child process after fork (see `ev_fork`).

`EV_CLEANUP`
The event loop is about to be destroyed (see `ev_cleanup`).

`EV_ASYNC`
The given async watcher has been asynchronously notified (see `ev_async`).

`EV_CUSTOM`
Not ever sent (or otherwise used) by libev itself, but can be freely used by libev users to signal watchers (e.g. via `ev_feed_event`).

`EV_ERROR`
An unspecified error has occurred, the watcher has been stopped. This might happen because the watcher could not be properly started because libev ran out of memory, a file descriptor was found to be closed or any other problem. Libev considers these application bugs.

You best act on it by reporting the problem and somehow coping with the watcher being stopped. Note that well-written programs should not receive an error ever, so when your watcher receives it, this usually indicates a bug in your program.

Libev will usually signal a few ‘‘dummy’’ events together with an error, for example it might indicate that a fd is readable or writable, and if your callbacks is well-written it can just attempt the operation and cope with the error from **read()** or **write()**. This will not work in multi-threaded programs, though, as the fd could already be closed and reused for another thing, so beware.

GENERIC WATCHER FUNCTIONS

`ev_init` (ev_TYPE *watcher, callback)
This macro initialises the generic portion of a watcher. The contents of the watcher object can be arbitrary (so `malloc` will do). Only the generic parts of the watcher are initialised, you *need* to call the type-specific `ev_TYPE_set` macro afterwards to initialise the type-specific parts. For each type there is also a `ev_TYPE_init` macro which rolls both calls into one.

You can reinitialise a watcher at any time as long as it has been stopped (or never started) and there are no pending events outstanding.

The callback is always of type `void (*)(struct ev_loop *loop, ev_TYPE *watcher, int revents)`.

Example: Initialise an `ev_io` watcher in two steps.

```
ev_io w;
ev_init (&w, my_cb);
ev_io_set (&w, STDIN_FILENO, EV_READ);
```

ev_TYPE_set (ev_TYPE *watcher, [args])
This macro initialises the type-specific parts of a watcher. You need to call ev_init at least once before you call this macro, but you can call ev_TYPE_set any number of times. You must not, however, call this macro on a watcher that is active (it can be pending, however, which is a difference to the ev_init macro).

Although some watcher types do not have type-specific arguments (e.g. ev_prepare) you still need to call its set macro.

See ev_init, above, for an example.

ev_TYPE_init (ev_TYPE *watcher, callback, [args])
This convenience macro rolls both ev_init and ev_TYPE_set macro calls into a single call. This is the most convenient method to initialise a watcher. The same limitations apply, of course.

Example: Initialise and set an ev_io watcher in one step.

```
ev_io_init (&w, my_cb, STDIN_FILENO, EV_READ);
```

ev_TYPE_start (loop, ev_TYPE *watcher)
Starts (activates) the given watcher. Only active watchers will receive events. If the watcher is already active nothing will happen.

Example: Start the ev_io watcher that is being abused as example in this whole section.

```
ev_io_start (EV_DEFAULT_UC, &w);
```

ev_TYPE_stop (loop, ev_TYPE *watcher)
Stops the given watcher if active, and clears the pending status (whether the watcher was active or not).

It is possible that stopped watchers are pending – for example, non-repeating timers are being stopped when they become pending – but calling ev_TYPE_stop ensures that the watcher is neither active nor pending. If you want to free or reuse the memory used by the watcher it is therefore a good idea to always call its ev_TYPE_stop function.

bool ev_is_active (ev_TYPE *watcher)
Returns a true value iff the watcher is active (i.e. it has been started and not yet been stopped). As long as a watcher is active you must not modify it.

bool ev_is_pending (ev_TYPE *watcher)
Returns a true value iff the watcher is pending, (i.e. it has outstanding events but its callback has not yet been invoked). As long as a watcher is pending (but not active) you must not call an init function on it (but ev_TYPE_set is safe), you must not change its priority, and you must make sure the watcher is available to libev (e.g. you cannot free () it).

callback ev_cb (ev_TYPE *watcher)
Returns the callback currently set on the watcher.

ev_set_cb (ev_TYPE *watcher, callback)
Change the callback. You can change the callback at virtually any time (modulo threads).

ev_set_priority (ev_TYPE *watcher, int priority)
int ev_priority (ev_TYPE *watcher)
Set and query the priority of the watcher. The priority is a small integer between EV_MAXPRI (default: 2) and EV_MINPRI (default: -2). Pending watchers with higher priority will be invoked before watchers with lower priority, but priority will not keep watchers from being executed (except for ev_idle watchers).

If you need to suppress invocation when higher priority events are pending you need to look at

`ev_idle` watchers, which provide this functionality.

You *must not* change the priority of a watcher as long as it is active or pending.

Setting a priority outside the range of `EV_MINPRI` to `EV_MAXPRI` is fine, as long as you do not mind that the priority value you query might or might not have been clamped to the valid range.

The default priority used by watchers when no priority has been set is always `0`, which is supposed to not be too high and not be too low :).

See ''WATCHER PRIORITY MODELS'', below, for a more thorough treatment of priorities.

ev_invoke (loop, ev_TYPE *watcher, int revents)
Invoke the `watcher` with the given `loop` and `revents`. Neither `loop` nor `revents` need to be valid as long as the watcher callback can deal with that fact, as both are simply passed through to the callback.

int ev_clear_pending (loop, ev_TYPE *watcher)
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`.

Sometimes it can be useful to ''poll'' a watcher instead of waiting for its callback to be invoked, which can be accomplished with this function.

ev_feed_event (loop, ev_TYPE *watcher, int revents)
Feeds the given event set into the event loop, as if the specified event had happened for the specified watcher (which must be a pointer to an initialised but not necessarily started event watcher). Obviously you must not free the watcher as long as it has pending events.

Stopping the watcher, letting libev invoke it, or calling `ev_clear_pending` will clear the pending event, even if the watcher was not started in the first place.

See also `ev_feed_fd_event` and `ev_feed_signal_event` for related functions that do not need a watcher.

See also the ''ASSOCIATING CUSTOM DATA WITH A WATCHER'' and ''BUILDING YOUR OWN COMPOSITE WATCHERS'' idioms.

WATCHER STATES

There are various watcher states mentioned throughout this manual – active, pending and so on. In this section these states and the rules to transition between them will be described in more detail – and while these rules might look complicated, they usually do ''the right thing''.

initialised
Before a watcher can be registered with the event loop it has to be initialised. This can be done with a call to `ev_TYPE_init`, or calls to `ev_init` followed by the watcher-specific `ev_TYPE_set` function.

In this state it is simply some block of memory that is suitable for use in an event loop. It can be moved around, freed, reused etc. at will – as long as you either keep the memory contents intact, or call `ev_TYPE_init` again.

started/running/active
Once a watcher has been started with a call to `ev_TYPE_start` it becomes property of the event loop, and is actively waiting for events. While in this state it cannot be accessed (except in a few documented ways), moved, freed or anything else – the only legal thing is to keep a pointer to it, and call libev functions on it that are documented to work on active watchers.

pending
If a watcher is active and libev determines that an event it is interested in has occurred (such as a timer expiring), it will become pending. It will stay in this pending state until either it is stopped or its callback is about to be invoked, so it is not normally pending inside the watcher callback.

The watcher might or might not be active while it is pending (for example, an expired non-repeating

timer can be pending but no longer active). If it is stopped, it can be freely accessed (e.g. by calling `ev_TYPE_set`), but it is still property of the event loop at this time, so cannot be moved, freed or reused. And if it is active the rules described in the previous item still apply.

It is also possible to feed an event on a watcher that is not active (e.g. via `ev_feed_event`), in which case it becomes pending without being active.

stopped

A watcher can be stopped implicitly by libev (in which case it might still be pending), or explicitly by calling its `ev_TYPE_stop` function. The latter will clear any pending state the watcher might be in, regardless of whether it was active or not, so stopping a watcher explicitly before freeing it is often a good idea.

While stopped (and not pending) the watcher is essentially in the initialised state, that is, it can be reused, moved, modified in any way you wish (but when you trash the memory block, you need to `ev_TYPE_init` it again).

WATCHER PRIORITY MODELS

Many event loops support *watcher priorities*, which are usually small integers that influence the ordering of event callback invocation between watchers in some way, all else being equal.

In libev, Watcher priorities can be set using `ev_set_priority`. See its description for the more technical details such as the actual priority range.

There are two common ways how these these priorities are being interpreted by event loops:

In the more common lock-out model, higher priorities "lock out" invocation of lower priority watchers, which means as long as higher priority watchers receive events, lower priority watchers are not being invoked.

The less common only-for-ordering model uses priorities solely to order callback invocation within a single event loop iteration: Higher priority watchers are invoked before lower priority ones, but they all get invoked before polling for new events.

Libev uses the second (only-for-ordering) model for all its watchers except for idle watchers (which use the lock-out model).

The rationale behind this is that implementing the lock-out model for watchers is not well supported by most kernel interfaces, and most event libraries will just poll for the same events again and again as long as their callbacks have not been executed, which is very inefficient in the common case of one high-priority watcher locking out a mass of lower priority ones.

Static (ordering) priorities are most useful when you have two or more watchers handling the same resource: a typical usage example is having an `ev_io` watcher to receive data, and an associated `ev_timer` to handle timeouts. Under load, data might be received while the program handles other jobs, but since timers normally get invoked first, the timeout handler will be executed before checking for data. In that case, giving the timer a lower priority than the I/O watcher ensures that I/O will be handled first even under adverse conditions (which is usually, but not always, what you want).

Since idle watchers use the "lock-out" model, meaning that idle watchers will only be executed when no same or higher priority watchers have received events, they can be used to implement the "lock-out" model when required.

For example, to emulate how many other event libraries handle priorities, you can associate an `ev_idle` watcher to each such watcher, and in the normal watcher callback, you just start the idle watcher. The real processing is done in the idle watcher callback. This causes libev to continuously poll and process kernel event data for the watcher, but when the lock-out case is known to be rare (which in turn is rare :), this is workable.

Usually, however, the lock-out model implemented that way will perform miserably under the type of load it was designed to handle. In that case, it might be preferable to stop the real watcher before starting the idle watcher, so the kernel will not have to process the event in case the actual processing will be delayed for considerable time.

Here is an example of an I/O watcher that should run at a strictly lower priority than the default, and which should only process data when no other events are pending:

```
ev_idle idle; // actual processing watcher
ev_io io;     // actual event watcher

static void
io_cb (EV_P_ ev_io *w, int revents)
{
  // stop the I/O watcher, we received the event, but
  // are not yet ready to handle it.
  ev_io_stop (EV_A_ w);

  // start the idle watcher to handle the actual event.
  // it will not be executed as long as other watchers
  // with the default priority are receiving events.
  ev_idle_start (EV_A_ &idle);
}

static void
idle_cb (EV_P_ ev_idle *w, int revents)
{
  // actual processing
  read (STDIN_FILENO, ...);

  // have to start the I/O watcher again, as
  // we have handled the event
  ev_io_start (EV_P_ &io);
}

// initialisation
ev_idle_init (&idle, idle_cb);
ev_io_init (&io, io_cb, STDIN_FILENO, EV_READ);
ev_io_start (EV_DEFAULT_ &io);
```

In the ''real'' world, it might also be beneficial to start a timer, so that low-priority connections can not be locked out forever under load. This enables your program to keep a lower latency for important connections during short periods of high load, while not completely locking out less important ones.

WATCHER TYPES

This section describes each watcher in detail, but will not repeat information given in the last section. Any initialisation/set macros, functions and members specific to the watcher type are explained.

Members are additionally marked with either *[read−only]*, meaning that, while the watcher is active, you can look at the member and expect some sensible content, but you must not modify it (you can modify it while the watcher is stopped to your hearts content), or *[read−write]*, which means you can expect it to have some sensible content while the watcher is active, but you can also modify it. Modifying it may not do something sensible or take immediate effect (or do anything at all), but libev will not crash or malfunction in any way.

`ev_io` – is this file descriptor readable or writable?

I/O watchers check whether a file descriptor is readable or writable in each iteration of the event loop, or, more precisely, when reading would not block the process and writing would at least be able to write some data. This behaviour is called level-triggering because you keep receiving events as long as the condition persists. Remember you can stop the watcher if you don't want to act on the event and neither want to receive future events.

In general you can register as many read and/or write event watchers per fd as you want (as long as you

don't confuse yourself). Setting all file descriptors to non-blocking mode is also usually a good idea (but not required if you know what you are doing).

Another thing you have to watch out for is that it is quite easy to receive ''spurious'' readiness notifications, that is, your callback might be called with `EV_READ` but a subsequent `read`(2) will actually block because there is no data. It is very easy to get into this situation even with a relatively standard program structure. Thus it is best to always use non-blocking I/O: An extra `read`(2) returning `EAGAIN` is far preferable to a program hanging until some data arrives.

If you cannot run the fd in non-blocking mode (for example you should not play around with an Xlib connection), then you have to separately re-test whether a file descriptor is really ready with a known-to-be good interface such as poll (fortunately in the case of Xlib, it already does this on its own, so its quite safe to use). Some people additionally use `SIGALRM` and an interval timer, just to be sure you won't block indefinitely.

But really, best use non-blocking mode.

The special problem of disappearing file descriptors

Some backends (e.g. kqueue, epoll, linuxaio) need to be told about closing a file descriptor (either due to calling `close` explicitly or any other means, such as `dup2`). The reason is that you register interest in some file descriptor, but when it goes away, the operating system will silently drop this interest. If another file descriptor with the same number then is registered with libev, there is no efficient way to see that this is, in fact, a different file descriptor.

To avoid having to explicitly tell libev about such cases, libev follows the following policy: Each time `ev_io_set` is being called, libev will assume that this is potentially a new file descriptor, otherwise it is assumed that the file descriptor stays the same. That means that you *have* to call `ev_io_set` (or `ev_io_init`) when you change the descriptor even if the file descriptor number itself did not change.

This is how one would do it normally anyway, the important point is that the libev application should not optimise around libev but should leave optimisations to libev.

The special problem of dup'ed file descriptors

Some backends (e.g. epoll), cannot register events for file descriptors, but only events for the underlying file descriptions. That means when you have `dup ()`'ed file descriptors or weirder constellations, and register events for them, only one file descriptor might actually receive events.

There is no workaround possible except not registering events for potentially `dup ()`'ed file descriptors, or to resort to `EVBACKEND_SELECT` or `EVBACKEND_POLL`.

The special problem of files

Many people try to use `select` (or libev) on file descriptors representing files, and expect it to become ready when their program doesn't block on disk accesses (which can take a long time on their own).

However, this cannot ever work in the ''expected'' way − you get a readiness notification as soon as the kernel knows whether and how much data is there, and in the case of open files, that's always the case, so you always get a readiness notification instantly, and your read (or possibly write) will still block on the disk I/O.

Another way to view it is that in the case of sockets, pipes, character devices and so on, there is another party (the sender) that delivers data on its own, but in the case of files, there is no such thing: the disk will not send data on its own, simply because it doesn't know what you wish to read − you would first have to request some data.

Since files are typically not-so-well supported by advanced notification mechanism, libev tries hard to emulate POSIX behaviour with respect to files, even though you should not use it. The reason for this is convenience: sometimes you want to watch STDIN or STDOUT, which is usually a tty, often a pipe, but also sometimes files or special devices (for example, `epoll` on Linux works with */dev/random* but not with */dev/urandom*), and even though the file might better be served with asynchronous I/O instead of with non-blocking I/O, it is still useful when it ''just works'' instead of freezing.

So avoid file descriptors pointing to files when you know it (e.g. use libeio), but use them when it is convenient, e.g. for STDIN/STDOUT, or when you rarely read from a file instead of from a socket, and want to reuse the same code path.

The special problem of fork

Some backends (epoll, kqueue, probably linuxaio) do not support `fork ()` at all or exhibit useless behaviour. Libev fully supports fork, but needs to be told about it in the child if you want to continue to use it in the child.

To support fork in your child processes, you have to call `ev_loop_fork ()` after a fork in the child, enable `EVFLAG_FORKCHECK`, or resort to `EVBACKEND_SELECT` or `EVBACKEND_POLL`.

The special problem of SIGPIPE

While not really specific to libev, it is easy to forget about `SIGPIPE`: when writing to a pipe whose other end has been closed, your program gets sent a SIGPIPE, which, by default, aborts your program. For most programs this is sensible behaviour, for daemons, this is usually undesirable.

So when you encounter spurious, unexplained daemon exits, make sure you ignore SIGPIPE (and maybe make sure you log the exit status of your daemon somewhere, as that would have given you a big clue).

*The special problem of **accept()**ing when you can't*

Many implementations of the POSIX `accept` function (for example, found in post−2004 Linux) have the peculiar behaviour of not removing a connection from the pending queue in all error cases.

For example, larger servers often run out of file descriptors (because of resource limits), causing `accept` to fail with `ENFILE` but not rejecting the connection, leading to libev signalling readiness on the next iteration again (the connection still exists after all), and typically causing the program to loop at 100% CPU usage.

Unfortunately, the set of errors that cause this issue differs between operating systems, there is usually little the app can do to remedy the situation, and no known thread-safe method of removing the connection to cope with overload is known (to me).

One of the easiest ways to handle this situation is to just ignore it − when the program encounters an overload, it will just loop until the situation is over. While this is a form of busy waiting, no OS offers an event-based way to handle this situation, so it's the best one can do.

A better way to handle the situation is to log any errors other than `EAGAIN` and `EWOULDBLOCK`, making sure not to flood the log with such messages, and continue as usual, which at least gives the user an idea of what could be wrong (''raise the ulimit!''). For extra points one could stop the `ev_io` watcher on the listening fd ''for a while'', which reduces CPU usage.

If your program is single-threaded, then you could also keep a dummy file descriptor for overload situations (e.g. by opening */dev/null*), and when you run into `ENFILE` or `EMFILE`, close it, run `accept`, close that fd, and create a new dummy fd. This will gracefully refuse clients under typical overload conditions.

The last way to handle it is to simply log the error and `exit`, as is often done with `malloc` failures, but this results in an easy opportunity for a DoS attack.

Watcher-Specific Functions

ev_io_init (ev_io *, callback, int fd, int events)
ev_io_set (ev_io *, int fd, int events)
: Configures an `ev_io` watcher. The `fd` is the file descriptor to receive events for and `events` is either `EV_READ`, `EV_WRITE` or `EV_READ | EV_WRITE`, to express the desire to receive the given events.

int fd [read−only]
: The file descriptor being watched.

int events [read−only]
The events being watched.

Examples

Example: Call `stdin_readable_cb` when STDIN_FILENO has become, well readable, but only once. Since it is likely line-buffered, you could attempt to read a whole line in the callback.

```
static void
stdin_readable_cb (struct ev_loop *loop, ev_io *w, int revents)
{
   ev_io_stop (loop, w);
  .. read from stdin here (or from w->fd) and handle any I/O errors
}

...
struct ev_loop *loop = ev_default_init (0);
ev_io stdin_readable;
ev_io_init (&stdin_readable, stdin_readable_cb, STDIN_FILENO, EV_READ);
ev_io_start (loop, &stdin_readable);
ev_run (loop, 0);
```

`ev_timer` **– relative and optionally repeating timeouts**

Timer watchers are simple relative timers that generate an event after a given time, and optionally repeating in regular intervals after that.

The timers are based on real time, that is, if you register an event that times out after an hour and you reset your system clock to January last year, it will still time out after (roughly) one hour. ‘‘Roughly’’ because detecting time jumps is hard, and some inaccuracies are unavoidable (the monotonic clock option helps a lot here).

The callback is guaranteed to be invoked only *after* its timeout has passed (not *at*, so on systems with very low-resolution clocks this might introduce a small delay, see ‘‘the special problem of being too early’’, below). If multiple timers become ready during the same loop iteration then the ones with earlier time-out values are invoked before ones of the same priority with later time-out values (but this is no longer true when a callback calls `ev_run` recursively).

Be smart about timeouts

Many real-world problems involve some kind of timeout, usually for error recovery. A typical example is an HTTP request – if the other side hangs, you want to raise some error after a while.

What follows are some ways to handle this problem, from obvious and inefficient to smart and efficient.

In the following, a 60 second activity timeout is assumed – a timeout that gets reset to 60 seconds each time there is activity (e.g. each time some data or other life sign was received).

1. Use a timer and stop, reinitialise and start it on activity.
   This is the most obvious, but not the most simple way: In the beginning, start the watcher:

   ```
   ev_timer_init (timer, callback, 60., 0.);
   ev_timer_start (loop, timer);
   ```

   Then, each time there is some activity, `ev_timer_stop` it, initialise it and start it again:

   ```
   ev_timer_stop (loop, timer);
   ev_timer_set (timer, 60., 0.);
   ev_timer_start (loop, timer);
   ```

   This is relatively simple to implement, but means that each time there is some activity, libev will first have to remove the timer from its internal data structure and then add it again. Libev tries to be fast, but it’s still not a constant-time operation.

2. Use a timer and re-start it with `ev_timer_again` inactivity.
   This is the easiest way, and involves using `ev_timer_again` instead of `ev_timer_start`.

   To implement this, configure an `ev_timer` with a `repeat` value of `60` and then call `ev_timer_again` at start and each time you successfully read or write some data. If you go into an idle state where you do not expect data to travel on the socket, you can `ev_timer_stop` the timer, and `ev_timer_again` will automatically restart it if need be.

   That means you can ignore both the `ev_timer_start` function and the `after` argument to `ev_timer_set`, and only ever use the `repeat` member and `ev_timer_again`.

   At start:

   ```
   ev_init (timer, callback);
   timer->repeat = 60.;
   ev_timer_again (loop, timer);
   ```

   Each time there is some activity:

   ```
   ev_timer_again (loop, timer);
   ```

   It is even possible to change the time-out on the fly, regardless of whether the watcher is active or not:

   ```
   timer->repeat = 30.;
   ev_timer_again (loop, timer);
   ```

   This is slightly more efficient then stopping/starting the timer each time you want to modify its timeout value, as libev does not have to completely remove and re-insert the timer from/into its internal data structure.

   It is, however, even simpler than the ‘‘obvious’’ way to do it.

3. Let the timer time out, but then re-arm it as required.
   This method is more tricky, but usually most efficient: Most timeouts are relatively long compared to the intervals between other activity – in our example, within 60 seconds, there are usually many I/O events with associated activity resets.

   In this case, it would be more efficient to leave the `ev_timer` alone, but remember the time of last activity, and check for a real timeout only within the callback:

   ```
   ev_tstamp timeout = 60.;
   ev_tstamp last_activity; // time of last activity
   ev_timer timer;

   static void
   callback (EV_P_ ev_timer *w, int revents)
   {
     // calculate when the timeout would happen
     ev_tstamp after = last_activity - ev_now (EV_A) + timeout;

     // if negative, it means we the timeout already occurred
     if (after < 0.)
       {
         // timeout occurred, take action
       }
     else
       {
         // callback was invoked, but there was some recent
         // activity. simply restart the timer to time out
         // after "after" seconds, which is the earliest time
         // the timeout can occur.
   ```

```
          ev_timer_set (w, after, 0.);
          ev_timer_start (EV_A_ w);
        }
    }
```

To summarise the callback: first calculate in how many seconds the timeout will occur (by calculating the absolute time when it would occur, `last_activity + timeout`, and subtracting the current time, `ev_now (EV_A)` from that).

If this value is negative, then we are already past the timeout, i.e. we timed out, and need to do whatever is needed in this case.

Otherwise, we now the earliest time at which the timeout would trigger, and simply start the timer with this timeout value.

In other words, each time the callback is invoked it will check whether the timeout occurred. If not, it will simply reschedule itself to check again at the earliest time it could time out. Rinse. Repeat.

This scheme causes more callback invocations (about one every 60 seconds minus half the average time between activity), but virtually no calls to libev to change the timeout.

To start the machinery, simply initialise the watcher and set `last_activity` to the current time (meaning there was some activity just now), then call the callback, which will "do the right thing" and start the timer:

```
last_activity = ev_now (EV_A);
ev_init (&timer, callback);
callback (EV_A_ &timer, 0);
```

When there is some activity, simply store the current time in `last_activity`, no libev calls at all:

```
if (activity detected)
  last_activity = ev_now (EV_A);
```

When your timeout value changes, then the timeout can be changed by simply providing a new value, stopping the timer and calling the callback, which will again do the right thing (for example, time out immediately :).

```
timeout = new_value;
ev_timer_stop (EV_A_ &timer);
callback (EV_A_ &timer, 0);
```

This technique is slightly more complex, but in most cases where the time-out is unlikely to be triggered, much more efficient.

4. Wee, just use a double-linked list for your timeouts.
   If there is not one request, but many thousands (millions...), all employing some kind of timeout with the same timeout value, then one can do even better:

   When starting the timeout, calculate the timeout value and put the timeout at the *end* of the list.

   Then use an `ev_timer` to fire when the timeout at the *beginning* of the list is expected to fire (for example, using the technique #3).

   When there is some activity, remove the timer from the list, recalculate the timeout, append it to the end of the list again, and make sure to update the `ev_timer` if it was taken from the beginning of the list.

   This way, one can manage an unlimited number of timeouts in O(1) time for starting, stopping and updating the timers, at the expense of a major complication, and having to use a constant timeout. The constant timeout ensures that the list stays sorted.

So which method the best?

Method #2 is a simple no-brain-required solution that is adequate in most situations. Method #3 requires a bit more thinking, but handles many cases better, and isn't very complicated either. In most case, choosing either one is fine, with #3 being better in typical situations.

Method #1 is almost always a bad idea, and buys you nothing. Method #4 is rather complicated, but extremely efficient, something that really pays off after the first million or so of active timers, i.e. it's usually overkill :)

The special problem of being too early

If you ask a timer to call your callback after three seconds, then you expect it to be invoked after three seconds − but of course, this cannot be guaranteed to infinite precision. Less obviously, it cannot be guaranteed to any precision by libev − imagine somebody suspending the process with a STOP signal for a few hours for example.

So, libev tries to invoke your callback as soon as possible *after* the delay has occurred, but cannot guarantee this.

A less obvious failure mode is calling your callback too early: many event loops compare timestamps with a ''elapsed delay >= requested delay'', but this can cause your callback to be invoked much earlier than you would expect.

To see why, imagine a system with a clock that only offers full second resolution (think windows if you can't come up with a broken enough OS yourself). If you schedule a one-second timer at the time 500.9, then the event loop will schedule your timeout to elapse at a system time of 500 (500.9 truncated to the resolution) + 1, or 501.

If an event library looks at the timeout 0.1s later, it will see ''501 >= 501'' and invoke the callback 0.1s after it was started, even though a one-second delay was requested − this is being ''too early'', despite best intentions.

This is the reason why libev will never invoke the callback if the elapsed delay equals the requested delay, but only when the elapsed delay is larger than the requested delay. In the example above, libev would only invoke the callback at system time 502, or 1.1s after the timer was started.

So, while libev cannot guarantee that your callback will be invoked exactly when requested, it *can* and *does* guarantee that the requested delay has actually elapsed, or in other words, it always errs on the ''too late'' side of things.

The special problem of time updates

Establishing the current time is a costly operation (it usually takes at least one system call): EV therefore updates its idea of the current time only before and after `ev_run` collects new events, which causes a growing difference between `ev_now ()` and `ev_time ()` when handling lots of events in one iteration.

The relative timeouts are calculated relative to the `ev_now ()` time. This is usually the right thing as this timestamp refers to the time of the event triggering whatever timeout you are modifying/starting. If you suspect event processing to be delayed and you *need* to base the timeout on the current time, use something like the following to adjust for it:

```
ev_timer_set (&timer, after + (ev_time () - ev_now ()), 0.);
```

If the event loop is suspended for a long time, you can also force an update of the time returned by `ev_now ()` by calling `ev_now_update ()`, although that will push the event time of all outstanding events further into the future.

The special problem of unsynchronised clocks

Modern systems have a variety of clocks − libev itself uses the normal ''wall clock'' clock and, if available, the monotonic clock (to avoid time jumps).

Neither of these clocks is synchronised with each other or any other clock on the system, so `ev_time ()` might return a considerably different time than `gettimeofday ()` or `time ()`. On a GNU/Linux system, for example, a call to `gettimeofday` might return a second count that is one higher than a directly following call to `time`.

The moral of this is to only compare libev-related timestamps with `ev_time ()` and `ev_now ()`, at least if you want better precision than a second or so.

One more problem arises due to this lack of synchronisation: if libev uses the system monotonic clock and you compare timestamps from `ev_time` or `ev_now` from when you started your timer and when your callback is invoked, you will find that sometimes the callback is a bit ''early''.

This is because `ev_timers` work in real time, not wall clock time, so libev makes sure your callback is not invoked before the delay happened, *measured according to the real time*, not the system clock.

If your timeouts are based on a physical timescale (e.g. ''time out this connection after 100 seconds'') then this shouldn't bother you as it is exactly the right behaviour.

If you want to compare wall clock/system timestamps to your timers, then you need to use `ev_periodics`, as these are based on the wall clock time, where your comparisons will always generate correct results.

The special problems of suspended animation

When you leave the server world it is quite customary to hit machines that can suspend/hibernate – what happens to the clocks during such a suspend?

Some quick tests made with a Linux 2.6.28 indicate that a suspend freezes all processes, while the clocks (`times`, `CLOCK_MONOTONIC`) continue to run until the system is suspended, but they will not advance while the system is suspended. That means, on resume, it will be as if the program was frozen for a few seconds, but the suspend time will not be counted towards `ev_timer` when a monotonic clock source is used. The real time clock advanced as expected, but if it is used as sole clocksource, then a long suspend would be detected as a time jump by libev, and timers would be adjusted accordingly.

I would not be surprised to see different behaviour in different between operating systems, OS versions or even different hardware.

The other form of suspend (job control, or sending a SIGSTOP) will see a time jump in the monotonic clocks and the realtime clock. If the program is suspended for a very long time, and monotonic clock sources are in use, then you can expect `ev_timers` to expire as the full suspension time will be counted towards the timers. When no monotonic clock source is in use, then libev will again assume a timejump and adjust accordingly.

It might be beneficial for this latter case to call `ev_suspend` and `ev_resume` in code that handles `SIGTSTP`, to at least get deterministic behaviour in this case (you can do nothing against `SIGSTOP`).

Watcher-Specific Functions and Data Members

ev_timer_init (ev_timer *, callback, ev_tstamp after, ev_tstamp repeat)
ev_timer_set (ev_timer *, ev_tstamp after, ev_tstamp repeat)

Configure the timer to trigger after `after` seconds (fractional and negative values are supported). If `repeat` is `0.`, then it will automatically be stopped once the timeout is reached. If it is positive, then the timer will automatically be configured to trigger again `repeat` seconds later, again, and again, until stopped manually.

The timer itself will do a best-effort at avoiding drift, that is, if you configure a timer to trigger every 10 seconds, then it will normally trigger at exactly 10 second intervals. If, however, your program cannot keep up with the timer (because it takes longer than those 10 seconds to do stuff) the timer will not fire more than once per event loop iteration.

ev_timer_again (loop, ev_timer *)

This will act as if the timer timed out, and restarts it again if it is repeating. It basically works like calling `ev_timer_stop`, updating the timeout to the `repeat` value and calling `ev_timer_start`.

The exact semantics are as in the following rules, all of which will be applied to the watcher:

If the timer is pending, the pending status is always cleared.
If the timer is started but non-repeating, stop it (as if it timed out, without invoking it).
If the timer is repeating, make the `repeat` value the new timeout and start the timer, if necessary.

This sounds a bit complicated, see ''Be smart about timeouts'', above, for a usage example.

ev_tstamp ev_timer_remaining (loop, ev_timer *)
Returns the remaining time until a timer fires. If the timer is active, then this time is relative to the current event loop time, otherwise it's the timeout value currently configured.

That is, after an `ev_timer_set (w, 5, 7)`, `ev_timer_remaining` returns `5`. When the timer is started and one second passes, `ev_timer_remaining` will return `4`. When the timer expires and is restarted, it will return roughly `7` (likely slightly less as callback invocation takes some time, too), and so on.

ev_tstamp repeat [read−write]
The current `repeat` value. Will be used each time the watcher times out or `ev_timer_again` is called, and determines the next timeout (if any), which is also when any modifications are taken into account.

Examples

Example: Create a timer that fires after 60 seconds.

```
static void
one_minute_cb (struct ev_loop *loop, ev_timer *w, int revents)
{
  .. one minute over, w is actually stopped right here
}

ev_timer mytimer;
ev_timer_init (&mytimer, one_minute_cb, 60., 0.);
ev_timer_start (loop, &mytimer);
```

Example: Create a timeout timer that times out after 10 seconds of inactivity.

```
static void
timeout_cb (struct ev_loop *loop, ev_timer *w, int revents)
{
  .. ten seconds without any activity
}

ev_timer mytimer;
ev_timer_init (&mytimer, timeout_cb, 0., 10.); /* note, only repeat used */
ev_timer_again (&mytimer); /* start timer */
ev_run (loop, 0);

// and in some piece of code that gets executed on any "activity":
// reset the timeout to start ticking again at 10 seconds
ev_timer_again (&mytimer);
```

`ev_periodic` **– to cron or not to cron?**
Periodic watchers are also timers of a kind, but they are very versatile (and unfortunately a bit complex).

Unlike `ev_timer`, periodic watchers are not based on real time (or relative time, the physical time that passes) but on wall clock time (absolute time, the thing you can read on your calendar or clock). The difference is that wall clock time can run faster or slower than real time, and time jumps are not uncommon (e.g. when you adjust your wrist-watch).

You can tell a periodic watcher to trigger after some specific point in time: for example, if you tell a periodic watcher to trigger ''in 10 seconds'' (by specifying e.g. `ev_now () + 10.`, that is, an absolute

time not a delay) and then reset your system clock to January of the previous year, then it will take a year or more to trigger the event (unlike an `ev_timer`, which would still trigger roughly 10 seconds after starting it, as it uses a relative timeout).

`ev_periodic` watchers can also be used to implement vastly more complex timers, such as triggering an event on each ‘‘midnight, local time’’, or other complicated rules. This cannot easily be done with `ev_timer` watchers, as those cannot react to time jumps.

As with timers, the callback is guaranteed to be invoked only when the point in time where it is supposed to trigger has passed. If multiple timers become ready during the same loop iteration then the ones with earlier time-out values are invoked before ones with later time-out values (but this is no longer true when a callback calls `ev_run` recursively).

Watcher-Specific Functions and Data Members

ev_periodic_init (ev_periodic *, callback, ev_tstamp offset, ev_tstamp interval, reschedule_cb)
ev_periodic_set (ev_periodic *, ev_tstamp offset, ev_tstamp interval, reschedule_cb)

Lots of arguments, let’s sort it out... There are basically three modes of operation, and we will explain them from simplest to most complex:

- absolute timer (offset = absolute time, interval = 0, reschedule_cb = 0)

  In this configuration the watcher triggers an event after the wall clock time `offset` has passed. It will not repeat and will not adjust when a time jump occurs, that is, if it is to be run at January 1st 2011 then it will be stopped and invoked when the system clock reaches or surpasses this point in time.

- repeating interval timer (offset = offset within interval, interval > 0, reschedule_cb = 0)

  In this mode the watcher will always be scheduled to time out at the next `offset + N * interval` time (for some integer N, which can also be negative) and then repeat, regardless of any time jumps. The `offset` argument is merely an offset into the `interval` periods.

  This can be used to create timers that do not drift with respect to the system clock, for example, here is an `ev_periodic` that triggers each hour, on the hour (with respect to UTC):

  ```
  ev_periodic_set (&periodic, 0., 3600., 0);
  ```

  This doesn’t mean there will always be 3600 seconds in between triggers, but only that the callback will be called when the system time shows a full hour (UTC), or more correctly, when the system time is evenly divisible by 3600.

  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 = offset (mod interval)`, regardless of any time jumps.

  The `interval` *MUST* be positive, and for numerical stability, the interval value should be higher than `1/8192` (which is around 100 microseconds) and `offset` should be higher than `0` and should have at most a similar magnitude as the current time (say, within a factor of ten). Typical values for offset are, in fact, `0` or something between `0` and `interval`, which is also the recommended range.

  Note also that there is an upper limit to how often a timer can fire (CPU speed for example), so if `interval` is very small then timing stability will of course deteriorate. Libev itself tries to be exact to be about one millisecond (if the OS supports it and the machine is fast enough).

- manual reschedule mode (offset ignored, interval ignored, reschedule_cb = callback)

  In this mode the values for `interval` and `offset` are both being ignored. Instead, each time the periodic watcher gets scheduled, the reschedule callback will be called with the watcher as first, and the current time as second argument.

  NOTE: *This callback MUST NOT stop or destroy any periodic watcher, ever, or make ANY other event loop modifications whatsoever, unless explicitly allowed by documentation here*.

If you need to stop it, return `now + 1e30` (or so, fudge fudge) and stop it afterwards (e.g. by starting an `ev_prepare` watcher, which is the only event loop modification you are allowed to do).

The callback prototype is `ev_tstamp (*reschedule_cb)(ev_periodic *w, ev_tstamp now)`, e.g.:

```
static ev_tstamp
my_rescheduler (ev_periodic *w, ev_tstamp now)
{
  return now + 60.;
}
```

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

NOTE: *This callback must always return a time that is higher than or equal to the passed `now` value*.

This can be used to create very complex timers, such as a timer that triggers on ‘‘next midnight, local time’’. To do this, you would calculate the next midnight after `now` and return the timestamp value for this. Here is a (completely untested, no error checking) example on how to do this:

```
#include <time.h>

static ev_tstamp
my_rescheduler (ev_periodic *w, ev_tstamp now)
{
  time_t tnow = (time_t)now;
  struct tm tm;
  localtime_r (&tnow, &tm);

  tm.tm_sec = tm.tm_min = tm.tm_hour = 0; // midnight current day
  ++tm.tm_mday; // midnight next day

  return mktime (&tm);
}
```

Note: this code might run into trouble on days that have more then two midnights (beginning and end).

ev_periodic_again (loop, ev_periodic *)

Simply stops and restarts the periodic watcher again. This is only useful when you changed some parameters or the reschedule callback would return a different time than the last time it was called (e.g. in a crond like program when the crontabs have changed).

ev_tstamp ev_periodic_at (ev_periodic *)

When active, returns the absolute time that the watcher is supposed to trigger next. This is not the same as the `offset` argument to `ev_periodic_set`, but indeed works even in interval and manual rescheduling modes.

ev_tstamp offset [read−write]

When repeating, this contains the offset value, otherwise this is the absolute point in time (the `offset` value passed to `ev_periodic_set`, although libev might modify this value for better numerical stability).

Can be modified any time, but changes only take effect when the periodic timer fires or `ev_periodic_again` is being called.

ev_tstamp interval [read−write]
: The current interval value. Can be modified any time, but changes only take effect when the periodic timer fires or `ev_periodic_again` is being called.

ev_tstamp (*reschedule_cb)(ev_periodic *w, ev_tstamp now) [read−write]
: The current reschedule callback, or `0`, if this functionality is switched off. Can be changed any time, but changes only take effect when the periodic timer fires or `ev_periodic_again` is being called.

Examples

Example: Call a callback every hour, or, more precisely, whenever the system time is divisible by 3600. The callback invocation times have potentially a lot of jitter, but good long-term stability.

```
static void
clock_cb (struct ev_loop *loop, ev_periodic *w, int revents)
{
  ... its now a full hour (UTC, or TAI or whatever your clock follows)
}

ev_periodic hourly_tick;
ev_periodic_init (&hourly_tick, clock_cb, 0., 3600., 0);
ev_periodic_start (loop, &hourly_tick);
```

Example: The same as above, but use a reschedule callback to do it:

```
#include <math.h>

static ev_tstamp
my_scheduler_cb (ev_periodic *w, ev_tstamp now)
{
  return now + (3600. - fmod (now, 3600.));
}

ev_periodic_init (&hourly_tick, clock_cb, 0., 0., my_scheduler_cb);
```

Example: Call a callback every hour, starting now:

```
ev_periodic hourly_tick;
ev_periodic_init (&hourly_tick, clock_cb,
                  fmod (ev_now (loop), 3600.), 3600., 0);
ev_periodic_start (loop, &hourly_tick);
```

`ev_signal` **– signal me when a signal gets signalled!**

Signal watchers will trigger an event when the process receives a specific signal one or more times. Even though signals are very asynchronous, libev will try its best to deliver signals synchronously, i.e. as part of the normal event processing, like any other event.

If you want signals to be delivered truly asynchronously, just use `sigaction` as you would do without libev and forget about sharing the signal. You can even use `ev_async` from a signal handler to synchronously wake up an event loop.

You can configure as many watchers as you like for the same signal, but only within the same loop, i.e. you can watch for `SIGINT` in your default loop and for `SIGIO` in another loop, but you cannot watch for `SIGINT` in both the default loop and another loop at the same time. At the moment, `SIGCHLD` is permanently tied to the default loop.

Only after the first watcher for a signal is started will libev actually register something with the kernel. It thus coexists with your own signal handlers as long as you don't register any with libev for the same signal.

If possible and supported, libev will install its handlers with `SA_RESTART` (or equivalent) behaviour enabled, so system calls should not be unduly interrupted. If you have a problem with system calls getting interrupted by signals you can block all signals in an `ev_check` watcher and unblock them in an

ev_prepare watcher.

The special problem of inheritance over fork/execve/pthread_create

Both the signal mask (sigprocmask) and the signal disposition (sigaction) are unspecified after starting a signal watcher (and after stopping it again), that is, libev might or might not block the signal, and might or might not set or restore the installed signal handler (but see EVFLAG_NOSIGMASK).

While this does not matter for the signal disposition (libev never sets signals to SIG_IGN, so handlers will be reset to SIG_DFL on execve), this matters for the signal mask: many programs do not expect certain signals to be blocked.

This means that before calling exec (from the child) you should reset the signal mask to whatever "default" you expect (all clear is a good choice usually).

The simplest way to ensure that the signal mask is reset in the child is to install a fork handler with pthread_atfork that resets it. That will catch fork calls done by libraries (such as the libc) as well.

In current versions of libev, the signal will not be blocked indefinitely unless you use the signalfd API (EV_SIGNALFD). While this reduces the window of opportunity for problems, it will not go away, as libev *has* to modify the signal mask, at least temporarily.

So I can't stress this enough: *If you do not reset your signal mask when you expect it to be empty, you have a race condition in your code*. This is not a libev-specific thing, this is true for most event libraries.

The special problem of threads signal handling

POSIX threads has problematic signal handling semantics, specifically, a lot of functionality (sigfd, sigwait etc.) only really works if all threads in a process block signals, which is hard to achieve.

When you want to use sigwait (or mix libev signal handling with your own for the same signals), you can tackle this problem by globally blocking all signals before creating any threads (or creating them with a fully set sigprocmask) and also specifying the EVFLAG_NOSIGMASK when creating loops. Then designate one thread as "signal receiver thread" which handles these signals. You can pass on any signals that libev might be interested in by calling ev_feed_signal.

Watcher-Specific Functions and Data Members

ev_signal_init (ev_signal *, callback, int signum)
ev_signal_set (ev_signal *, int signum)
: Configures the watcher to trigger on the given signal number (usually one of the SIGxxx constants).

int signum [read−only]
: The signal the watcher watches out for.

Examples

Example: Try to exit cleanly on SIGINT.

```
static void
sigint_cb (struct ev_loop *loop, ev_signal *w, int revents)
{
  ev_break (loop, EVBREAK_ALL);
}

ev_signal signal_watcher;
ev_signal_init (&signal_watcher, sigint_cb, SIGINT);
ev_signal_start (loop, &signal_watcher);
```

ev_child **– watch out for process status changes**

Child watchers trigger when your process receives a SIGCHLD in response to some child status changes (most typically when a child of yours dies or exits). It is permissible to install a child watcher *after* the child has been forked (which implies it might have already exited), as long as the event loop isn't entered (or is continued from a watcher), i.e., forking and then immediately registering a watcher for the child is fine, but forking and registering a watcher a few event loop iterations later or in the next callback invocation is not.

Only the default event loop is capable of handling signals, and therefore you can only register child watchers in the default event loop.

Due to some design glitches inside libev, child watchers will always be handled at maximum priority (their priority is set to `EV_MAXPRI` by libev)

Process Interaction

Libev grabs `SIGCHLD` as soon as the default event loop is initialised. This is necessary to guarantee proper behaviour even if the first child watcher is started after the child exits. The occurrence of `SIGCHLD` is recorded asynchronously, but child reaping is done synchronously as part of the event loop processing. Libev always reaps all children, even ones not watched.

Overriding the Built-In Processing

Libev offers no special support for overriding the built-in child processing, but if your application collides with libev's default child handler, you can override it easily by installing your own handler for `SIGCHLD` after initialising the default loop, and making sure the default loop never gets destroyed. You are encouraged, however, to use an event-based approach to child reaping and thus use libev's support for that, so other libev users can use `ev_child` watchers freely.

Stopping the Child Watcher

Currently, the child watcher never gets stopped, even when the child terminates, so normally one needs to stop the watcher in the callback. Future versions of libev might stop the watcher automatically when a child exit is detected (calling `ev_child_stop` twice is not a problem).

Watcher-Specific Functions and Data Members

ev_child_init (ev_child *, callback, int pid, int trace)
ev_child_set (ev_child *, int pid, int trace)
: Configures the watcher to wait for status changes of process `pid` (or *any* process if `pid` is specified as `0`). The callback can look at the `rstatus` member of the `ev_child` watcher structure to see the status word (use the macros from `sys/wait.h` and see your systems `waitpid` documentation). The `rpid` member contains the pid of the process causing the status change. `trace` must be either `0` (only activate the watcher when the process terminates) or `1` (additionally activate the watcher when the process is stopped or continued).

int pid [read−only]
: The process id this watcher watches out for, or `0`, meaning any process id.

int rpid [read−write]
: The process id that detected a status change.

int rstatus [read−write]
: The process exit/trace status caused by `rpid` (see your systems `waitpid` and `sys/wait.h` documentation for details).

Examples

Example: `fork()` a new process and install a child handler to wait for its completion.

```
ev_child cw;

static void
child_cb (EV_P_ ev_child *w, int revents)
{
  ev_child_stop (EV_A_ w);
  printf ("process %d exited with status %x\n", w->rpid, w->rstatus);
}

pid_t pid = fork ();
```

```
        if (pid < 0)
          // error
        else if (pid == 0)
          {
            // the forked child executes here
            exit (1);
          }
        else
          {
            ev_child_init (&cw, child_cb, pid, 0);
            ev_child_start (EV_DEFAULT_ &cw);
          }
```

`ev_stat` **– did the file attributes just change?**

This watches a file system path for attribute changes. That is, it calls `stat` on that path in regular intervals (or when the OS says it changed) and sees if it changed compared to the last time, invoking the callback if it did. Starting the watcher `stat`'s the file, so only changes that happen after the watcher has been started will be reported.

The path does not need to exist: changing from ''path exists'' to ''path does not exist'' is a status change like any other. The condition ''path does not exist'' (or more correctly ''path cannot be stat'ed'') is signified by the `st_nlink` field being zero (which is otherwise always forced to be at least one) and all the other fields of the stat buffer having unspecified contents.

The path *must not* end in a slash or contain special components such as `.` or `...`. The path *should* be absolute: If it is relative and your working directory changes, then the behaviour is undefined.

Since there is no portable change notification interface available, the portable implementation simply calls `stat(2)` regularly on the path to see if it changed somehow. You can specify a recommended polling interval for this case. If you specify a polling interval of `0` (highly recommended!) then a *suitable, unspecified default* value will be used (which you can expect to be around five seconds, although this might change dynamically). Libev will also impose a minimum interval which is currently around `0.1`, but that's usually overkill.

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

At the time of this writing, the only OS-specific interface implemented is the Linux inotify interface (implementing kqueue support is left as an exercise for the reader. Note, however, that the author sees no way of implementing `ev_stat` semantics with kqueue, except as a hint).

ABI Issues (Largefile Support)

Libev by default (unless the user overrides this) uses the default compilation environment, which means that on systems with large file support disabled by default, you get the 32 bit version of the stat structure. When using the library from programs that change the ABI to use 64 bit file offsets the programs will fail. In that case you have to compile libev with the same flags to get binary compatibility. This is obviously the case with any flags that change the ABI, but the problem is most noticeably displayed with ev_stat and large file support.

The solution for this is to lobby your distribution maker to make large file interfaces available by default (as e.g. FreeBSD does) and not optional. Libev cannot simply switch on large file support because it has to exchange stat structures with application programs compiled using the default compilation environment.

Inotify and Kqueue

When `inotify (7)` support has been compiled into libev and present at runtime, it will be used to speed up change detection where possible. The inotify descriptor will be created lazily when the first `ev_stat` watcher is being started.

Inotify presence does not change the semantics of `ev_stat` watchers except that changes might be detected earlier, and in some cases, to avoid making regular `stat` calls. Even in the presence of inotify

support there are many cases where libev has to resort to regular `stat` polling, but as long as kernel 2.6.25 or newer is used (2.6.24 and older have too many bugs), the path exists (i.e. stat succeeds), and the path resides on a local filesystem (libev currently assumes only ext2/3, jfs, reiserfs and xfs are fully working) libev usually gets away without polling.

There is no support for kqueue, as apparently it cannot be used to implement this functionality, due to the requirement of having a file descriptor open on the object at all times, and detecting renames, unlinks etc. is difficult.

`stat ()` is a synchronous operation

Libev doesn't normally do any kind of I/O itself, and so is not blocking the process. The exception are `ev_stat` watchers – those call `stat ()`, which is a synchronous operation.

For local paths, this usually doesn't matter: unless the system is very busy or the intervals between stat's are large, a stat call will be fast, as the path data is usually in memory already (except when starting the watcher).

For networked file systems, calling `stat ()` can block an indefinite time due to network issues, and even under good conditions, a stat call often takes multiple milliseconds.

Therefore, it is best to avoid using `ev_stat` watchers on networked paths, although this is fully supported by libev.

The special problem of stat time resolution

The `stat ()` system call only supports full-second resolution portably, and even on systems where the resolution is higher, most file systems still only support whole seconds.

That means that, if the time is the only thing that changes, you can easily miss updates: on the first update, `ev_stat` detects a change and calls your callback, which does something. When there is another update within the same second, `ev_stat` will be unable to detect unless the stat data does change in other ways (e.g. file size).

The solution to this is to delay acting on a change for slightly more than a second (or till slightly after the next full second boundary), using a roughly one-second-delay `ev_timer` (e.g. `ev_timer_set (w, 0., 1.02); ev_timer_again (loop, w)`).

The `.02` offset is added to work around small timing inconsistencies of some operating systems (where the second counter of the current time might be be delayed. One such system is the Linux kernel, where a call to `gettimeofday` might return a timestamp with a full second later than a subsequent `time` call – if the equivalent of `time ()` is used to update file times then there will be a small window where the kernel uses the previous second to update file times but libev might already execute the timer callback).

Watcher-Specific Functions and Data Members

ev_stat_init (ev_stat *, callback, const char *path, ev_tstamp interval)
ev_stat_set (ev_stat *, const char *path, ev_tstamp interval)
: Configures the watcher to wait for status changes of the given `path`. The `interval` is a hint on how quickly a change is expected to be detected and should normally be specified as `0` to let libev choose a suitable value. The memory pointed to by `path` must point to the same path for as long as the watcher is active.

  The callback will receive an `EV_STAT` event when a change was detected, relative to the attributes at the time the watcher was started (or the last change was detected).

ev_stat_stat (loop, ev_stat *)
: Updates the stat buffer immediately with new values. If you change the watched path in your callback, you could call this function to avoid detecting this change (while introducing a race condition if you are not the only one changing the path). Can also be useful simply to find out the new values.

ev_statdata attr [read−only]
: The most-recently detected attributes of the file. Although the type is `ev_statdata`, this is usually the (or one of the) `struct stat` types suitable for your system, but you can only rely on the

POSIX-standardised members to be present. If the `st_nlink` member is `0`, then there was some error while `stat`ing the file.

ev_statdata prev [read−only]
: The previous attributes of the file. The callback gets invoked whenever `prev != attr`, or, more precisely, one or more of these members differ: `st_dev`, `st_ino`, `st_mode`, `st_nlink`, `st_uid`, `st_gid`, `st_rdev`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`.

ev_tstamp interval [read−only]
: The specified interval.

const char *path [read−only]
: The file system path that is being watched.

Examples

Example: Watch `/etc/passwd` for attribute changes.

```
static void
passwd_cb (struct ev_loop *loop, ev_stat *w, int revents)
{
  /* /etc/passwd changed in some way */
  if (w->attr.st_nlink)
    {
      printf ("passwd current size  %ld\n", (long)w->attr.st_size);
      printf ("passwd current atime %ld\n", (long)w->attr.st_mtime);
      printf ("passwd current mtime %ld\n", (long)w->attr.st_mtime);
    }
  else
    /* you shalt not abuse printf for puts */
    puts ("wow, /etc/passwd is not there, expect problems. "
          "if this is windows, they already arrived\n");
}

...
ev_stat passwd;

ev_stat_init (&passwd, passwd_cb, "/etc/passwd", 0.);
ev_stat_start (loop, &passwd);
```

Example: Like above, but additionally use a one-second delay so we do not miss updates (however, frequent updates will delay processing, too, so one might do the work both on `ev_stat` callback invocation *and* on `ev_timer` callback invocation).

```
static ev_stat passwd;
static ev_timer timer;

static void
timer_cb (EV_P_ ev_timer *w, int revents)
{
  ev_timer_stop (EV_A_ w);

  /* now it's one second after the most recent passwd change */
}

static void
stat_cb (EV_P_ ev_stat *w, int revents)
{
  /* reset the one-second timer */
```

```
  ev_timer_again (EV_A_ &timer);
}

...
ev_stat_init (&passwd, stat_cb, "/etc/passwd", 0.);
ev_stat_start (loop, &passwd);
ev_timer_init (&timer, timer_cb, 0., 1.02);
```

`ev_idle` **– when you've got nothing better to do...**

Idle watchers trigger events when no other events of the same or higher priority are pending (prepare, check and other idle watchers do not count as receiving "events").

That is, as long as your process is busy handling sockets or timeouts (or even signals, imagine) of the same or higher priority it will not be triggered. But when your process is idle (or only lower-priority watchers are pending), the idle watchers are being called once per event loop iteration – until stopped, that is, or your process receives more events and becomes busy again with higher priority stuff.

The most noteworthy effect is that as long as any idle watchers are active, the process will not block when waiting for new events.

Apart from keeping your process non-blocking (which is a useful effect on its own sometimes), idle watchers are a good place to do "pseudo-background processing", or delay processing stuff to after the event loop has handled all outstanding events.

Abusing an `ev_idle` watcher for its side-effect

As long as there is at least one active idle watcher, libev will never sleep unnecessarily. Or in other words, it will loop as fast as possible. For this to work, the idle watcher doesn't need to be invoked at all – the lowest priority will do.

This mode of operation can be useful together with an `ev_check` watcher, to do something on each event loop iteration – for example to balance load between different connections.

See "Abusing an ev_check watcher for its side-effect" for a longer example.

Watcher-Specific Functions and Data Members

ev_idle_init (ev_idle *, callback)

Initialises and configures the idle watcher – it has no parameters of any kind. There is a `ev_idle_set` macro, but using it is utterly pointless, believe me.

Examples

Example: Dynamically allocate an `ev_idle` watcher, start it, and in the callback, free it. Also, use no error checking, as usual.

```
static void
idle_cb (struct ev_loop *loop, ev_idle *w, int revents)
{
  // stop the watcher
  ev_idle_stop (loop, w);

  // now we can free it
  free (w);

  // now do something you wanted to do when the program has
  // no longer anything immediate to do.
}

ev_idle *idle_watcher = malloc (sizeof (ev_idle));
ev_idle_init (idle_watcher, idle_cb);
ev_idle_start (loop, idle_watcher);
```

`ev_prepare` and `ev_check` – customise your event loop!

Prepare and check watchers are often (but not always) used in pairs: prepare watchers get invoked before the process blocks and check watchers afterwards.

You *must not* call `ev_run` (or similar functions that enter the current event loop) or `ev_loop_fork` from either `ev_prepare` or `ev_check` watchers. Other loops than the current one are fine, however. The rationale behind this is that you do not need to check for recursion in those watchers, i.e. the sequence will always be `ev_prepare`, blocking, `ev_check` so if you have one watcher of each kind they will always be called in pairs bracketing the blocking call.

Their main purpose is to integrate other event mechanisms into libev and their use is somewhat advanced. They could be used, for example, to track variable changes, implement your own watchers, integrate net-snmp or a coroutine library and lots more. They are also occasionally useful if you cache some data and want to flush it before blocking (for example, in X programs you might want to do an `XFlush ()` in an `ev_prepare` watcher).

This is done by examining in each prepare call which file descriptors need to be watched by the other library, registering `ev_io` watchers for them and starting an `ev_timer` watcher for any timeouts (many libraries provide exactly this functionality). Then, in the check watcher, you check for any events that occurred (by checking the pending status of all watchers and stopping them) and call back into the library. The I/O and timer callbacks will never actually be called (but must be valid nevertheless, because you never know, you know?).

As another example, the Perl Coro module uses these hooks to integrate coroutines into libev programs, by yielding to other active coroutines during each prepare and only letting the process block if no coroutines are ready to run (it's actually more complicated: it only runs coroutines with priority higher than or equal to the event loop and one coroutine of lower priority, but only once, using idle watchers to keep the event loop from blocking if lower-priority coroutines are active, thus mapping low-priority coroutines to idle/background tasks).

When used for this purpose, it is recommended to give `ev_check` watchers highest (`EV_MAXPRI`) priority, to ensure that they are being run before any other watchers after the poll (this doesn't matter for `ev_prepare` watchers).

Also, `ev_check` watchers (and `ev_prepare` watchers, too) should not activate ("feed") events into libev. While libev fully supports this, they might get executed before other `ev_check` watchers did their job. As `ev_check` watchers are often used to embed other (non-libev) event loops those other event loops might be in an unusable state until their `ev_check` watcher ran (always remind yourself to coexist peacefully with others).

Abusing an `ev_check` watcher for its side-effect

`ev_check` (and less often also `ev_prepare`) watchers can also be useful because they are called once per event loop iteration. For example, if you want to handle a large number of connections fairly, you normally only do a bit of work for each active connection, and if there is more work to do, you wait for the next event loop iteration, so other connections have a chance of making progress.

Using an `ev_check` watcher is almost enough: it will be called on the next event loop iteration. However, that isn't as soon as possible – without external events, your `ev_check` watcher will not be invoked.

This is where `ev_idle` watchers come in handy – all you need is a single global idle watcher that is active as long as you have one active `ev_check` watcher. The `ev_idle` watcher makes sure the event loop will not sleep, and the `ev_check` watcher makes sure a callback gets invoked. Neither watcher alone can do that.

Watcher-Specific Functions and Data Members

ev_prepare_init (ev_prepare *, callback)
ev_check_init (ev_check *, callback)

Initialises and configures the prepare or check watcher – they have no parameters of any kind. There are `ev_prepare_set` and `ev_check_set` macros, but using them is utterly, utterly, utterly and completely pointless.

Examples

There are a number of principal ways to embed other event loops or modules into libev. Here are some ideas on how to include libadns into libev (there is a Perl module named `EV::ADNS` that does this, which you could use as a working example. Another Perl module named `EV::Glib` embeds a Glib main context into libev, and finally, `Glib::EV` embeds EV into the Glib event loop).

Method 1: Add IO watchers and a timeout watcher in a prepare handler, and in a check watcher, destroy them and call into libadns. What follows is pseudo-code only of course. This requires you to either use a low priority for the check watcher or use `ev_clear_pending` explicitly, as the callbacks for the IO/timeout watchers might not have been called yet.

```
static ev_io iow [nfd];
static ev_timer tw;

static void
io_cb (struct ev_loop *loop, ev_io *w, int revents)
{
}

// create io watchers for each fd and a timer before blocking
static void
adns_prepare_cb (struct ev_loop *loop, ev_prepare *w, int revents)
{
  int timeout = 3600000;
  struct pollfd fds [nfd];
  // actual code will need to loop here and realloc etc.
  adns_beforepoll (ads, fds, &nfd, &timeout, timeval_from (ev_time ()));

  /* the callback is illegal, but won't be called as we stop during check */
  ev_timer_init (&tw, 0, timeout * 1e-3, 0.);
  ev_timer_start (loop, &tw);

  // create one ev_io per pollfd
  for (int i = 0; i < nfd; ++i)
    {
      ev_io_init (iow + i, io_cb, fds [i].fd,
        ((fds [i].events & POLLIN ? EV_READ : 0)
         | (fds [i].events & POLLOUT ? EV_WRITE : 0)));

      fds [i].revents = 0;
      ev_io_start (loop, iow + i);
    }
}

// stop all watchers after blocking
static void
adns_check_cb (struct ev_loop *loop, ev_check *w, int revents)
{
  ev_timer_stop (loop, &tw);

  for (int i = 0; i < nfd; ++i)
    {
      // set the relevant poll flags
      // could also call adns_processreadable etc. here
      struct pollfd *fd = fds + i;
```

```
        int revents = ev_clear_pending (iow + i);
        if (revents & EV_READ ) fd->revents |= fd->events & POLLIN;
        if (revents & EV_WRITE) fd->revents |= fd->events & POLLOUT;

        // now stop the watcher
        ev_io_stop (loop, iow + i);
      }

    adns_afterpoll (adns, fds, nfd, timeval_from (ev_now (loop));
  }
```

Method 2: This would be just like method 1, but you run `adns_afterpoll` in the prepare watcher and would dispose of the check watcher.

Method 3: If the module to be embedded supports explicit event notification (libadns does), you can also make use of the actual watcher callbacks, and only destroy/create the watchers in the prepare watcher.

```
static void
timer_cb (EV_P_ ev_timer *w, int revents)
{
  adns_state ads = (adns_state)w->data;
  update_now (EV_A);

  adns_processtimeouts (ads, &tv_now);
}

static void
io_cb (EV_P_ ev_io *w, int revents)
{
  adns_state ads = (adns_state)w->data;
  update_now (EV_A);

  if (revents & EV_READ ) adns_processreadable  (ads, w->fd, &tv_now);
  if (revents & EV_WRITE) adns_processwriteable (ads, w->fd, &tv_now);
}

// do not ever call adns_afterpoll
```

Method 4: Do not use a prepare or check watcher because the module you want to embed is not flexible enough to support it. Instead, you can override their poll function. The drawback with this solution is that the main loop is now no longer controllable by EV. The `Glib::EV` module uses this approach, effectively embedding EV as a client into the horrible libglib event loop.

```
static gint
event_poll_func (GPollFD *fds, guint nfds, gint timeout)
{
  int got_events = 0;

  for (n = 0; n < nfds; ++n)
    // create/start io watcher that sets the relevant bits in fds[n] and incre

  if (timeout >= 0)
    // create/start timer

  // poll
  ev_run (EV_A_ 0);
```

```
          // stop timer again
          if (timeout >= 0)
            ev_timer_stop (EV_A_ &to);

          // stop io watchers again - their callbacks should have set
          for (n = 0; n < nfds; ++n)
            ev_io_stop (EV_A_ iow [n]);

          return got_events;
        }
```

`ev_embed` **– when one backend isn't enough...**

This is a rather advanced watcher type that lets you embed one event loop into another (currently only `ev_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).

There are primarily two reasons you would want that: work around bugs and prioritise I/O.

As an example for a bug workaround, the kqueue backend might only support sockets on some platform, so it is unusable as generic backend, but you still want to make use of it because you have many sockets and it scales so nicely. In this case, you would create a kqueue-based loop and embed it into your default loop (which might use e.g. poll). Overall operation will be a bit slower because first libev has to call `poll` and then `kevent`, but at least you can use both mechanisms for what they are best: `kqueue` for scalable sockets and `poll` if you want it to work :)

As for prioritising I/O: under rare circumstances you have the case where some fds have to be watched and handled very quickly (with low latency), and even priorities and idle watchers might have too much overhead. In this case you would put all the high priority stuff in one loop and all the rest in a second one, and embed the second one in the first.

As long as the watcher is active, the callback will be invoked every time there might be events pending in the embedded loop. The callback must then call `ev_embed_sweep (mainloop, watcher)` to make a single sweep and invoke their callbacks (the callback doesn't need to invoke the `ev_embed_sweep` function directly, it could also start an idle watcher to give the embedded loop strictly lower priority for example).

You can also set the callback to `0`, in which case the embed watcher will automatically execute the embedded loop sweep whenever necessary.

Fork detection will be handled transparently while the `ev_embed` watcher is active, i.e., the embedded loop will automatically be forked when the embedding loop forks. In other cases, the user is responsible for calling `ev_loop_fork` on the embedded loop.

Unfortunately, not all backends are embeddable: only the ones returned by `ev_embeddable_backends` are, which, unfortunately, does not include any portable one.

So when you want to use this feature you will always have to be prepared that you cannot get an embeddable loop. The recommended way to get around this is to have a separate variables for your embeddable loop, try to create it, and if that fails, use the normal loop for everything.

`ev_embed` and fork

While the `ev_embed` watcher is running, forks in the embedding loop will automatically be applied to the embedded loop as well, so no special fork handling is required in that case. When the watcher is not running, however, it is still the task of the libev user to call `ev_loop_fork ()` as applicable.

Watcher-Specific Functions and Data Members

ev_embed_init (ev_embed *, callback, struct ev_loop *embedded_loop)
ev_embed_set (ev_embed *, struct ev_loop *embedded_loop)

Configures the watcher to embed the given loop, which must be embeddable. If the callback is `0`, then `ev_embed_sweep` will be invoked automatically, otherwise it is the responsibility of the callback to

invoke it (it will continue to be called until the sweep has been done, if you do not want that, you need to temporarily stop the embed watcher).

ev_embed_sweep (loop, ev_embed *)
Make a single, non-blocking sweep over the embedded loop. This works similarly to `ev_run (embedded_loop, EVRUN_NOWAIT)`, but in the most appropriate way for embedded loops.

struct ev_loop *other [read−only]
The embedded event loop.

Examples

Example: Try to get an embeddable event loop and embed it into the default event loop. If that is not possible, use the default loop. The default loop is stored in `loop_hi`, while the embeddable loop is stored in `loop_lo` (which is `loop_hi` in the case no embeddable loop can be used).

```
struct ev_loop *loop_hi = ev_default_init (0);
struct ev_loop *loop_lo = 0;
ev_embed embed;

// see if there is a chance of getting one that works
// (remember that a flags value of 0 means autodetection)
loop_lo = ev_embeddable_backends () & ev_recommended_backends ()
  ? ev_loop_new (ev_embeddable_backends () & ev_recommended_backends ())
  : 0;

// if we got one, then embed it, otherwise default to loop_hi
if (loop_lo)
  {
    ev_embed_init (&embed, 0, loop_lo);
    ev_embed_start (loop_hi, &embed);
  }
else
  loop_lo = loop_hi;
```

Example: Check if kqueue is available but not recommended and create a kqueue backend for use with sockets (which usually work with any kqueue implementation). Store the kqueue/socket−only event loop in `loop_socket`. (One might optionally use `EVFLAG_NOENV`, too).

```
struct ev_loop *loop = ev_default_init (0);
struct ev_loop *loop_socket = 0;
ev_embed embed;

if (ev_supported_backends () & ~ev_recommended_backends () & EVBACKEND_KQUEUE)
  if ((loop_socket = ev_loop_new (EVBACKEND_KQUEUE))
    {
      ev_embed_init (&embed, 0, loop_socket);
      ev_embed_start (loop, &embed);
    }

if (!loop_socket)
  loop_socket = loop;

// now use loop_socket for all sockets, and loop for everything else
```

`ev_fork` **– the audacity to resume the event loop after a fork**
Fork watchers are called when a `fork ()` was detected (usually because whoever is a good citizen cared to tell libev about it by calling `ev_loop_fork`). The invocation is done before the event loop blocks next and before `ev_check` watchers are being called, and only in the child after the fork. If whoever good

citizen calling `ev_default_fork` cheats and calls it in the wrong process, the fork handlers will be invoked, too, of course.

The special problem of life after fork - how is it possible?

Most uses of `fork()` consist of forking, then some simple calls to set up/change the process environment, followed by a call to `exec()`. This sequence should be handled by libev without any problems.

This changes when the application actually wants to do event handling in the child, or both parent in child, in effect "continuing" after the fork.

The default mode of operation (for libev, with application help to detect forks) is to duplicate all the state in the child, as would be expected when *either* the parent *or* the child process continues.

When both processes want to continue using libev, then this is usually the wrong result. In that case, usually one process (typically the parent) is supposed to continue with all watchers in place as before, while the other process typically wants to start fresh, i.e. without any active watchers.

The cleanest and most efficient way to achieve that with libev is to simply create a new event loop, which of course will be "empty", and use that for new watchers. This has the advantage of not touching more memory than necessary, and thus avoiding the copy-on-write, and the disadvantage of having to use multiple event loops (which do not support signal watchers).

When this is not possible, or you want to use the default loop for other reasons, then in the process that wants to start "fresh", call `ev_loop_destroy (EV_DEFAULT)` followed by `ev_default_loop (...)`. Destroying the default loop will "orphan" (not stop) all registered watchers, so you have to be careful not to execute code that modifies those watchers. Note also that in that case, you have to re-register any signal watchers.

Watcher-Specific Functions and Data Members

ev_fork_init (ev_fork *, callback)

Initialises and configures the fork watcher - it has no parameters of any kind. There is a `ev_fork_set` macro, but using it is utterly pointless, really.

`ev_cleanup` **- even the best things end**

Cleanup watchers are called just before the event loop is being destroyed by a call to `ev_loop_destroy`.

While there is no guarantee that the event loop gets destroyed, cleanup watchers provide a convenient method to install cleanup hooks for your program, worker threads and so on - you just to make sure to destroy the loop when you want them to be invoked.

Cleanup watchers are invoked in the same way as any other watcher. Unlike all other watchers, they do not keep a reference to the event loop (which makes a lot of sense if you think about it). Like all other watchers, you can call libev functions in the callback, except `ev_cleanup_start`.

Watcher-Specific Functions and Data Members

ev_cleanup_init (ev_cleanup *, callback)

Initialises and configures the cleanup watcher - it has no parameters of any kind. There is a `ev_cleanup_set` macro, but using it is utterly pointless, I assure you.

Example: Register an atexit handler to destroy the default loop, so any cleanup functions are called.

```
static void
program_exits (void)
{
  ev_loop_destroy (EV_DEFAULT_UC);
}

...
atexit (program_exits);
```

`ev_async` **– how to wake up an event loop**

In general, you cannot use an `ev_loop` from multiple threads or other asynchronous sources such as signal handlers (as opposed to multiple event loops – those are of course safe to use in different threads).

Sometimes, however, you need to wake up an event loop you do not control, for example because it belongs to another thread. This is what `ev_async` watchers do: as long as the `ev_async` watcher is active, you can signal it by calling `ev_async_send`, which is thread– and signal safe.

This functionality is very similar to `ev_signal` watchers, as signals, too, are asynchronous in nature, and signals, too, will be compressed (i.e. the number of callback invocations may be less than the number of `ev_async_send` calls). In fact, you could use signal watchers as a kind of ‘‘global async watchers’’ by using a watcher on an otherwise unused signal, and `ev_feed_signal` to signal this watcher from another thread, even without knowing which loop owns the signal.

Queueing

`ev_async` does not support queueing of data in any way. The reason is that the author does not know of a simple (or any) algorithm for a multiple-writer-single-reader queue that works in all cases and doesn't need elaborate support such as pthreads or unportable memory access semantics.

That means that if you want to queue data, you have to provide your own queue. But at least I can tell you how to implement locking around your queue:

queueing from a signal handler context

To implement race-free queueing, you simply add to the queue in the signal handler but you block the signal handler in the watcher callback. Here is an example that does that for some fictitious SIGUSR1 handler:

```
static ev_async mysig;

static void
sigusr1_handler (void)
{
  sometype data;

  // no locking etc.
  queue_put (data);
  ev_async_send (EV_DEFAULT_ &mysig);
}

static void
mysig_cb (EV_P_ ev_async *w, int revents)
{
  sometype data;
  sigset_t block, prev;

  sigemptyset (&block);
  sigaddset (&block, SIGUSR1);
  sigprocmask (SIG_BLOCK, &block, &prev);

  while (queue_get (&data))
    process (data);

  if (sigismember (&prev, SIGUSR1)
    sigprocmask (SIG_UNBLOCK, &block, 0);
}
```

(Note: pthreads in theory requires you to use `pthread_setmask` instead of `sigprocmask` when you use threads, but libev doesn't do it either...).

queueing from a thread context

The strategy for threads is different, as you cannot (easily) block threads but you can easily preempt them, so to queue safely you need to employ a traditional mutex lock, such as in this pthread example:

```
static ev_async mysig;
static pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

static void
otherthread (void)
{
  // only need to lock the actual queueing operation
  pthread_mutex_lock (&mymutex);
  queue_put (data);
  pthread_mutex_unlock (&mymutex);

  ev_async_send (EV_DEFAULT_ &mysig);
}

static void
mysig_cb (EV_P_ ev_async *w, int revents)
{
  pthread_mutex_lock (&mymutex);

  while (queue_get (&data))
    process (data);

  pthread_mutex_unlock (&mymutex);
}
```

Watcher-Specific Functions and Data Members

ev_async_init (ev_async *, callback)

Initialises and configures the async watcher - it has no parameters of any kind. There is a `ev_async_set` macro, but using it is utterly pointless, trust me.

ev_async_send (loop, ev_async *)

Sends/signals/activates the given `ev_async` watcher, that is, feeds an `EV_ASYNC` event on the watcher into the event loop, and instantly returns.

Unlike `ev_feed_event`, this call is safe to do from other threads, signal or similar contexts (see the discussion of `EV_ATOMIC_T` in the embedding section below on what exactly this means).

Note that, as with other watchers in libev, multiple events might get compressed into a single callback invocation (another way to look at this is that `ev_async` watchers are level-triggered: they are set on `ev_async_send`, reset when the event loop detects that).

This call incurs the overhead of at most one extra system call per event loop iteration, if the event loop is blocked, and no syscall at all if the event loop (or your program) is processing events. That means that repeated calls are basically free (there is no need to avoid calls for performance reasons) and that the overhead becomes smaller (typically zero) under load.

bool = ev_async_pending (ev_async *)

Returns a non-zero value when `ev_async_send` has been called on the watcher but the event has not yet been processed (or even noted) by the event loop.

`ev_async_send` sets a flag in the watcher and wakes up the loop. When the loop iterates next and checks for the watcher to have become active, it will reset the flag again. `ev_async_pending` can be used to very quickly check whether invoking the loop might be a good idea.

Not that this does *not* check whether the watcher itself is pending, only whether it has been requested

to make this watcher pending: there is a time window between the event loop checking and resetting the async notification, and the callback being invoked.

OTHER FUNCTIONS

There are some other functions of possible interest. Described. Here. Now.

ev_once (loop, int fd, int events, ev_tstamp timeout, callback, arg)

This function combines a simple timer and an I/O watcher, calls your callback on whichever event happens first and automatically stops both watchers. This is useful if you want to wait for a single event on an fd or timeout without having to allocate/configure/start/stop/free one or more watchers yourself.

If `fd` is less than 0, then no I/O watcher will be started and the `events` argument is being ignored. Otherwise, an `ev_io` watcher for the given `fd` and `events` set will be created and started.

If `timeout` is less than 0, then no timeout watcher will be started. Otherwise an `ev_timer` watcher with after = `timeout` (and repeat = 0) will be started. `0` is a valid timeout.

The callback has the type `void (*cb)(int revents, void *arg)` and is passed an `revents` set like normal event callbacks (a combination of `EV_ERROR`, `EV_READ`, `EV_WRITE` or `EV_TIMER`) and the `arg` value passed to `ev_once`. Note that it is possible to receive *both* a timeout and an io event at the same time − you probably should give io events precedence.

Example: wait up to ten seconds for data to appear on STDIN_FILENO.

```
static void stdin_ready (int revents, void *arg)
{
  if (revents & EV_READ)
    /* stdin might have data for us, joy! */;
  else if (revents & EV_TIMER)
    /* doh, nothing entered */;
}

ev_once (STDIN_FILENO, EV_READ, 10., stdin_ready, 0);
```

ev_feed_fd_event (loop, int fd, int revents)

Feed an event on the given fd, as if a file descriptor backend detected the given events.

ev_feed_signal_event (loop, int signum)

Feed an event as if the given signal occurred. See also `ev_feed_signal`, which is async-safe.

COMMON OR USEFUL IDIOMS (OR BOTH)

This section explains some common idioms that are not immediately obvious. Note that examples are sprinkled over the whole manual, and this section only contains stuff that wouldn't fit anywhere else.

ASSOCIATING CUSTOM DATA WITH A WATCHER

Each watcher has, by default, a `void *data` member that you can read or modify at any time: libev will completely ignore it. This can be used to associate arbitrary data with your watcher. If you need more data and don't want to allocate memory separately and store a pointer to it in that data member, you can also "subclass" the watcher type and provide your own data:

```
struct my_io
{
  ev_io io;
  int otherfd;
  void *somedata;
  struct whatever *mostinteresting;
};

...
struct my_io w;
```

```
ev_io_init (&w.io, my_cb, fd, EV_READ);
```

And since your callback will be called with a pointer to the watcher, you can cast it back to your own type:

```
static void my_cb (struct ev_loop *loop, ev_io *w_, int revents)
{
  struct my_io *w = (struct my_io *)w_;
  ...
}
```

More interesting and less C-conformant ways of casting your callback function type instead have been omitted.

BUILDING YOUR OWN COMPOSITE WATCHERS

Another common scenario is to use some data structure with multiple embedded watchers, in effect creating your own watcher that combines multiple libev event sources into one "super-watcher":

```
struct my_biggy
{
  int some_data;
  ev_timer t1;
  ev_timer t2;
}
```

In this case getting the pointer to `my_biggy` is a bit more complicated: Either you store the address of your `my_biggy` struct in the `data` member of the watcher (for woozies or C++ coders), or you need to use some pointer arithmetic using `offsetof` inside your watchers (for real programmers):

```
#include <stddef.h>

static void
t1_cb (EV_P_ ev_timer *w, int revents)
{
  struct my_biggy big = (struct my_biggy *)
    (((char *)w) - offsetof (struct my_biggy, t1));
}

static void
t2_cb (EV_P_ ev_timer *w, int revents)
{
  struct my_biggy big = (struct my_biggy *)
    (((char *)w) - offsetof (struct my_biggy, t2));
}
```

AVOIDING FINISHING BEFORE RETURNING

Often you have structures like this in event-based programs:

```
callback ()
{
  free (request);
}

request = start_new_request (..., callback);
```

The intent is to start some "lengthy" operation. The `request` could be used to cancel the operation, or do other things with it.

It's not uncommon to have code paths in `start_new_request` that immediately invoke the callback, for example, to report errors. Or you add some caching layer that finds that it can skip the lengthy aspects of the operation and simply invoke the callback with the result.

The problem here is that this will happen *before* `start_new_request` has returned, so `request` is not

set.

Even if you pass the request by some safer means to the callback, you might want to do something to the request after starting it, such as canceling it, which probably isn't working so well when the callback has already been invoked.

A common way around all these issues is to make sure that `start_new_request` *always* returns before the callback is invoked. If `start_new_request` immediately knows the result, it can artificially delay invoking the callback by using a `prepare` or `idle` watcher for example, or more sneakily, by reusing an existing (stopped) watcher and pushing it into the pending queue:

```
ev_set_cb (watcher, callback);
ev_feed_event (EV_A_ watcher, 0);
```

This way, `start_new_request` can safely return before the callback is invoked, while not delaying callback invocation too much.

MODEL/NESTED EVENT LOOP INVOCATIONS AND EXIT CONDITIONS

Often (especially in GUI toolkits) there are places where you have *modal* interaction, which is most easily implemented by recursively invoking `ev_run`.

This brings the problem of exiting – a callback might want to finish the main `ev_run` call, but not the nested one (e.g. user clicked "Quit", but a modal "Are you sure?" dialog is still waiting), or just the nested one and not the main one (e.g. user clocked "Ok" in a modal dialog), or some other combination: In these cases, a simple `ev_break` will not work.

The solution is to maintain "break this loop" variable for each `ev_run` invocation, and use a loop around `ev_run` until the condition is triggered, using `EVRUN_ONCE`:

```
// main loop
int exit_main_loop = 0;

while (!exit_main_loop)
  ev_run (EV_DEFAULT_ EVRUN_ONCE);

// in a modal watcher
int exit_nested_loop = 0;

while (!exit_nested_loop)
  ev_run (EV_A_ EVRUN_ONCE);
```

To exit from any of these loops, just set the corresponding exit variable:

```
// exit modal loop
exit_nested_loop = 1;

// exit main program, after modal loop is finished
exit_main_loop = 1;

// exit both
exit_main_loop = exit_nested_loop = 1;
```

THREAD LOCKING EXAMPLE

Here is a fictitious example of how to run an event loop in a different thread from where callbacks are being invoked and watchers are created/added/removed.

For a real-world example, see the `EV::Loop::Async` perl module, which uses exactly this technique (which is suited for many high-level languages).

The example uses a pthread mutex to protect the loop data, a condition variable to wait for callback invocations, an async watcher to notify the event loop thread and an unspecified mechanism to wake up the main thread.

First, you need to associate some data with the event loop:

```
typedef struct {
  mutex_t lock; /* global loop lock */
  ev_async async_w;
  thread_t tid;
  cond_t invoke_cv;
} userdata;

void prepare_loop (EV_P)
{
   // for simplicity, we use a static userdata struct.
   static userdata u;

   ev_async_init (&u->async_w, async_cb);
   ev_async_start (EV_A_ &u->async_w);

   pthread_mutex_init (&u->lock, 0);
   pthread_cond_init (&u->invoke_cv, 0);

   // now associate this with the loop
   ev_set_userdata (EV_A_ u);
   ev_set_invoke_pending_cb (EV_A_ l_invoke);
   ev_set_loop_release_cb (EV_A_ l_release, l_acquire);

   // then create the thread running ev_run
   pthread_create (&u->tid, 0, l_run, EV_A);
}
```

The callback for the `ev_async` watcher does nothing: the watcher is used solely to wake up the event loop so it takes notice of any new watchers that might have been added:

```
static void
async_cb (EV_P_ ev_async *w, int revents)
{
   // just used for the side effects
}
```

The `l_release` and `l_acquire` callbacks simply unlock/lock the mutex protecting the loop data, respectively.

```
static void
l_release (EV_P)
{
  userdata *u = ev_userdata (EV_A);
  pthread_mutex_unlock (&u->lock);
}

static void
l_acquire (EV_P)
{
  userdata *u = ev_userdata (EV_A);
  pthread_mutex_lock (&u->lock);
}
```

The event loop thread first acquires the mutex, and then jumps straight into `ev_run`:

```
void *
l_run (void *thr_arg)
{
  struct ev_loop *loop = (struct ev_loop *)thr_arg;

  l_acquire (EV_A);
  pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, 0);
  ev_run (EV_A_ 0);
  l_release (EV_A);

  return 0;
}
```

Instead of invoking all pending watchers, the `l_invoke` callback will signal the main thread via some unspecified mechanism (signals? pipe writes? `Async::Interrupt`?) and then waits until all pending watchers have been called (in a while loop because a) spurious wakeups are possible and b) skipping inter-thread-communication when there are no pending watchers is very beneficial):

```
static void
l_invoke (EV_P)
{
  userdata *u = ev_userdata (EV_A);

  while (ev_pending_count (EV_A))
    {
      wake_up_other_thread_in_some_magic_or_not_so_magic_way ();
      pthread_cond_wait (&u->invoke_cv, &u->lock);
    }
}
```

Now, whenever the main thread gets told to invoke pending watchers, it will grab the lock, call `ev_invoke_pending` and then signal the loop thread to continue:

```
static void
real_invoke_pending (EV_P)
{
  userdata *u = ev_userdata (EV_A);

  pthread_mutex_lock (&u->lock);
  ev_invoke_pending (EV_A);
  pthread_cond_signal (&u->invoke_cv);
  pthread_mutex_unlock (&u->lock);
}
```

Whenever you want to start/stop a watcher or do other modifications to an event loop, you will now have to lock:

```
ev_timer timeout_watcher;
userdata *u = ev_userdata (EV_A);

ev_timer_init (&timeout_watcher, timeout_cb, 5.5, 0.);

pthread_mutex_lock (&u->lock);
ev_timer_start (EV_A_ &timeout_watcher);
ev_async_send (EV_A_ &u->async_w);
pthread_mutex_unlock (&u->lock);
```

Note that sending the `ev_async` watcher is required because otherwise an event loop currently blocking in the kernel will have no knowledge about the newly added timer. By waking up the loop it will pick up

any new watchers in the next event loop iteration.

THREADS, COROUTINES, CONTINUATIONS, QUEUES... INSTEAD OF CALLBACKS

While the overhead of a callback that e.g. schedules a thread is small, it is still an overhead. If you embed libev, and your main usage is with some kind of threads or coroutines, you might want to customise libev so that doesn't need callbacks anymore.

Imagine you have coroutines that you can switch to using a function `switch_to (coro)`, that libev runs in a coroutine called `libev_coro` and that due to some magic, the currently active coroutine is stored in a global called `current_coro`. Then you can build your own "wait for libev event" primitive by changing `EV_CB_DECLARE` and `EV_CB_INVOKE` (note the differing `;` conventions):

```
#define EV_CB_DECLARE(type)   struct my_coro *cb;
#define EV_CB_INVOKE(watcher) switch_to ((watcher)->cb)
```

That means instead of having a C callback function, you store the coroutine to switch to in each watcher, and instead of having libev call your callback, you instead have it switch to that coroutine.

A coroutine might now wait for an event with a function called `wait_for_event`. (the watcher needs to be started, as always, but it doesn't matter when, or whether the watcher is active or not when this function is called):

```
void
wait_for_event (ev_watcher *w)
{
  ev_set_cb (w, current_coro);
  switch_to (libev_coro);
}
```

That basically suspends the coroutine inside `wait_for_event` and continues the libev coroutine, which, when appropriate, switches back to this or any other coroutine.

You can do similar tricks if you have, say, threads with an event queue – instead of storing a coroutine, you store the queue object and instead of switching to a coroutine, you push the watcher onto the queue and notify any waiters.

To embed libev, see "EMBEDDING", but in short, it's easiest to create two files, *my_ev.h* and *my_ev.c* that include the respective libev files:

```
// my_ev.h
#define EV_CB_DECLARE(type)   struct my_coro *cb;
#define EV_CB_INVOKE(watcher) switch_to ((watcher)->cb)
#include "../libev/ev.h"

// my_ev.c
#define EV_H "my_ev.h"
#include "../libev/ev.c"
```

And then use *my_ev.h* when you would normally use *ev.h*, and compile *my_ev.c* into your project. When properly specifying include paths, you can even use *ev.h* as header file name directly.

LIBEVENT EMULATION

Libev offers a compatibility emulation layer for libevent. It cannot emulate the internals of libevent, so here are some usage hints:

- Only the libevent−1.4.1−beta API is being emulated.

  This was the newest libevent version available when libev was implemented, and is still mostly unchanged in 2010.

- Use it by including <event.h>, as usual.

- The following members are fully supported: ev_base, ev_callback, ev_arg, ev_fd, ev_res, ev_events.

- Avoid using ev_flags and the EVLIST_*-macros, while it is maintained by libev, it does not work exactly the same way as in libevent (consider it a private API).
- Priorities are not currently supported. Initialising priorities will fail and all watchers will have the same priority, even though there is an ev_pri field.
- In libevent, the last base created gets the signals, in libev, the base that registered the signal gets the signals.
- Other members are not supported.
- The libev emulation is *not* ABI compatible to libevent, you need to use the libev header file and library.

C++ SUPPORT

C API

The normal C API should work fine when used from C++: both ev.h and the libev sources can be compiled as C++. Therefore, code that uses the C API will work fine.

Proper exception specifications might have to be added to callbacks passed to libev: exceptions may be thrown only from watcher callbacks, all other callbacks (allocator, syserr, loop acquire/release and periodic reschedule callbacks) must not throw exceptions, and might need a `noexcept` specification. If you have code that needs to be compiled as both C and C++ you can use the `EV_NOEXCEPT` macro for this:

```
static void
fatal_error (const char *msg) EV_NOEXCEPT
{
  perror (msg);
  abort ();
}

...
ev_set_syserr_cb (fatal_error);
```

The only API functions that can currently throw exceptions are `ev_run`, `ev_invoke`, `ev_invoke_pending` and `ev_loop_destroy` (the latter because it runs cleanup watchers).

Throwing exceptions in watcher callbacks is only supported if libev itself is compiled with a C++ compiler or your C and C++ environments allow throwing exceptions through C libraries (most do).

C++ API

Libev comes with some simplistic wrapper classes for C++ that mainly allow you to use some convenience methods to start/stop watchers and also change the callback model to a model using method callbacks on objects.

To use it,

```
#include <ev++.h>
```

This automatically includes *ev.h* and puts all of its definitions (many of them macros) into the global namespace. All C++ specific things are put into the `ev` namespace. It should support all the same embedding options as *ev.h*, most notably `EV_MULTIPLICITY`.

Care has been taken to keep the overhead low. The only data member the C++ classes add (compared to plain C-style watchers) is the event loop pointer that the watcher is associated with (or no additional members at all if you disable `EV_MULTIPLICITY` when embedding libev).

Currently, functions, static and non-static member functions and classes with `operator ()` can be used as callbacks. Other types should be easy to add as long as they only need one additional pointer for context. If you need support for other types of functors please contact the author (preferably after implementing it).

For all this to work, your C++ compiler either has to use the same calling conventions as your C compiler (for static member functions), or you have to embed libev and compile libev itself as C++.

Here is a list of things available in the `ev` namespace:

`ev::READ`, `ev::WRITE` etc.
These are just enum values with the same values as the `EV_READ` etc. macros from *ev.h*.

`ev::tstamp`, `ev::now`
Aliases to the same types/functions as with the `ev_` prefix.

`ev::io`, `ev::timer`, `ev::periodic`, `ev::idle`, `ev::sig` etc.
For each `ev_TYPE` watcher in *ev.h* there is a corresponding class of the same name in the `ev` namespace, with the exception of `ev_signal` which is called `ev::sig` to avoid clashes with the `signal` macro defined by many implementations.

All of those classes have these methods:

ev::TYPE::TYPE ()
ev::TYPE::TYPE (loop)
ev::TYPE::~TYPE
The constructor (optionally) takes an event loop to associate the watcher with. If it is omitted, it will use `EV_DEFAULT`.

The constructor calls `ev_init` for you, which means you have to call the `set` method before starting it.

It will not set a callback, however: You have to call the templated `set` method to set a callback before you can start the watcher.

(The reason why you have to use a method is a limitation in C++ which does not allow explicit template arguments for constructors).

The destructor automatically stops the watcher if it is active.

w->set<class, &class::method> (object *)
This method sets the callback method to call. The method has to have a signature of `void (*)(ev_TYPE &, int)`, it receives the watcher as first argument and the `revents` as second. The object must be given as parameter and is stored in the `data` member of the watcher.

This method synthesizes efficient thunking code to call your method from the C callback that libev requires. If your compiler can inline your callback (i.e. it is visible to it at the place of the `set` call and your compiler is good :), then the method will be fully inlined into the thunking function, making it as fast as a direct C callback.

Example: simple class declaration and watcher initialisation

```
struct myclass
{
  void io_cb (ev::io &w, int revents) { }
}

myclass obj;
ev::io iow;
iow.set <myclass, &myclass::io_cb> (&obj);
```

w->set (object *)
This is a variation of a method callback - leaving out the method to call will default the method to `operator ()`, which makes it possible to use functor objects without having to manually specify the `operator ()` all the time. Incidentally, you can then also leave out the template argument list.

The `operator ()` method prototype must be `void operator ()(watcher &w, int revents)`.

See the method-`set` above for more details.

Example: use a functor object as callback.

```
struct myfunctor
{
  void operator() (ev::io &w, int revents)
  {
    ...
  }
}

myfunctor f;

ev::io w;
w.set (&f);
```

w->set<function> (void *data = 0)
Also sets a callback, but uses a static method or plain function as callback. The optional `data` argument will be stored in the watcher's `data` member and is free for you to use.

The prototype of the `function` must be `void (*)(ev::TYPE &w, int)`.

See the method-`set` above for more details.

Example: Use a plain function as callback.

```
static void io_cb (ev::io &w, int revents) { }
iow.set <io_cb> ();
```

w->set (loop)
Associates a different `struct ev_loop` with this watcher. You can only do this when the watcher is inactive (and not pending either).

w->set ([arguments])
Basically the same as `ev_TYPE_set` (except for `ev::embed` watchers>), with the same arguments. Either this method or a suitable start method must be called at least once. Unlike the C counterpart, an active watcher gets automatically stopped and restarted when reconfiguring it with this method.

For `ev::embed` watchers this method is called `set_embed`, to avoid clashing with the `set (loop)` method.

w->start ()
Starts the watcher. Note that there is no `loop` argument, as the constructor already stores the event loop.

w->start ([arguments])
Instead of calling `set` and `start` methods separately, it is often convenient to wrap them in one call. Uses the same type of arguments as the configure `set` method of the watcher.

w->stop ()
Stops the watcher if it is active. Again, no `loop` argument.

w->again () (`ev::timer`, `ev::periodic` only)
For `ev::timer` and `ev::periodic`, this invokes the corresponding `ev_TYPE_again` function.

w->sweep () (`ev::embed` only)
Invokes `ev_embed_sweep`.

w->update () (`ev::stat` only)
Invokes `ev_stat_stat`.

Example: Define a class with two I/O and idle watchers, start the I/O watchers in the constructor.

```
class myclass
{
  ev::io   io  ; void io_cb   (ev::io   &w, int revents);
  ev::io   io2 ; void io2_cb  (ev::io   &w, int revents);
  ev::idle idle; void idle_cb (ev::idle &w, int revents);

  myclass (int fd)
  {
    io  .set <myclass, &myclass::io_cb  > (this);
    io2 .set <myclass, &myclass::io2_cb > (this);
    idle.set <myclass, &myclass::idle_cb> (this);

    io.set (fd, ev::WRITE); // configure the watcher
    io.start ();            // start it whenever convenient

    io2.start (fd, ev::READ); // set + start in one call
  }
};
```

OTHER LANGUAGE BINDINGS

Libev does not offer other language bindings itself, but bindings for a number of languages exist in the form of third-party packages. If you know any interesting language binding in addition to the ones listed here, drop me a note.

Perl

The EV module implements the full libev API and is actually used to test libev. EV is developed together with libev. Apart from the EV core module, there are additional modules that implement libev-compatible interfaces to `libadns` (`EV::ADNS`, but `AnyEvent::DNS` is preferred nowadays), `Net::SNMP` (`Net::SNMP::EV`) and the `libglib` event core (`Glib::EV` and `EV::Glib`).

It can be found and installed via CPAN, its homepage is at <http://software.schmorp.de/pkg/EV>.

Python

Python bindings can be found at <http://code.google.com/p/pyev/>. It seems to be quite complete and well-documented.

Ruby

Tony Arcieri has written a ruby extension that offers access to a subset of the libev API and adds file handle abstractions, asynchronous DNS and more on top of it. It can be found via gem servers. Its homepage is at <http://rev.rubyforge.org/>.

Roger Pack reports that using the link order `-lws2_32 -lmsvcrt-ruby-190` makes rev work even on mingw.

Haskell

A haskell binding to libev is available at <http://hackage.haskell.org/cgi-bin/hackage-scripts/package/hlibev>.

D

Leandro Lucarella has written a D language binding (*ev.d*) for libev, to be found at <http://www.llucax.com.ar/proj/ev.d/index.html>.

Ocaml

Erkki Seppala has written Ocaml bindings for libev, to be found at <http://modeemi.cs.tut.fi/~flux/software/ocaml-ev/>.

Lua

Brian Maher has written a partial interface to libev for lua (at the time of this writing, only `ev_io` and `ev_timer`), to be found at <http://github.com/brimworks/lua-ev>.

Javascript
: Node.js (<http://nodejs.org>) uses libev as the underlying event library.

Others
: There are others, and I stopped counting.

MACRO MAGIC

Libev can be compiled with a variety of options, the most fundamental of which is `EV_MULTIPLICITY`. This option determines whether (most) functions and callbacks have an initial `struct ev_loop *` argument.

To make it easier to write programs that cope with either variant, the following macros are defined:

`EV_A`, `EV_A_`
: This provides the loop *argument* for functions, if one is required (“ev loop argument”). The `EV_A` form is used when this is the sole argument, `EV_A_` is used when other arguments are following. Example:

```
ev_unref (EV_A);
ev_timer_add (EV_A_ watcher);
ev_run (EV_A_ 0);
```

It assumes the variable `loop` of type `struct ev_loop *` is in scope, which is often provided by the following macro.

`EV_P`, `EV_P_`
: This provides the loop *parameter* for functions, if one is required (“ev loop parameter”). The `EV_P` form is used when this is the sole parameter, `EV_P_` is used when other parameters are following. Example:

```
// this is how ev_unref is being declared
static void ev_unref (EV_P);

// this is how you can declare your typical callback
static void cb (EV_P_ ev_timer *w, int revents)
```

It declares a parameter `loop` of type `struct ev_loop *`, quite suitable for use with `EV_A`.

`EV_DEFAULT`, `EV_DEFAULT_`
: Similar to the other two macros, this gives you the value of the default loop, if multiple loops are supported (“ev loop default”). The default loop will be initialised if it isn’t already initialised.

For non-multiplicity builds, these macros do nothing, so you always have to initialise the loop somewhere.

`EV_DEFAULT_UC`, `EV_DEFAULT_UC_`
: Usage identical to `EV_DEFAULT` and `EV_DEFAULT_`, but requires that the default loop has been initialised (`UC` == unchecked). Their behaviour is undefined when the default loop has not been initialised by a previous execution of `EV_DEFAULT`, `EV_DEFAULT_` or `ev_default_init (...)`.

It is often prudent to use `EV_DEFAULT` when initialising the first watcher in a function but use `EV_DEFAULT_UC` afterwards.

Example: Declare and initialise a check watcher, utilising the above macros so it will work regardless of whether multiple loops are supported or not.

```
static void
check_cb (EV_P_ ev_timer *w, int revents)
{
  ev_check_stop (EV_A_ w);
}
```

```
ev_check check;
ev_check_init (&check, check_cb);
ev_check_start (EV_DEFAULT_ &check);
ev_run (EV_DEFAULT_ 0);
```

EMBEDDING

Libev can (and often is) directly embedded into host applications. Examples of applications that embed it include the Deliantra Game Server, the EV perl module, the GNU Virtual Private Ethernet (gvpe) and rxvt-unicode.

The goal is to enable you to just copy the necessary files into your source directory without having to change even a single line in them, so you can easily upgrade by simply copying (or having a checked-out copy of libev somewhere in your source tree).

FILESETS

Depending on what features you need you need to include one or more sets of files in your application.

CORE EVENT LOOP

To include only the libev core (all the `ev_*` functions), with manual configuration (no autoconf):

```
#define EV_STANDALONE 1
#include "ev.c"
```

This will automatically include *ev.h*, too, and should be done in a single C source file only to provide the function implementations. To use it, do the same for *ev.h* in all files wishing to use this API (best done by writing a wrapper around *ev.h* that you can include instead and where you can put other configuration options):

```
#define EV_STANDALONE 1
#include "ev.h"
```

Both header files and implementation files can be compiled with a C++ compiler (at least, that's a stated goal, and breakage will be treated as a bug).

You need the following files in your source tree, or in a directory in your include path (e.g. in libev/ when using -Ilibev):

```
ev.h
ev.c
ev_vars.h
ev_wrap.h

ev_win32.c      required on win32 platforms only

ev_select.c     only when select backend is enabled
ev_poll.c       only when poll backend is enabled
ev_epoll.c      only when the epoll backend is enabled
ev_linuxaio.c   only when the linux aio backend is enabled
ev_kqueue.c     only when the kqueue backend is enabled
ev_port.c       only when the solaris port backend is enabled
```

ev.c includes the backend files directly when enabled, so you only need to compile this single file.

LIBEVENT COMPATIBILITY API

To include the libevent compatibility API, also include:

```
#include "event.c"
```

in the file including *ev.c*, and:

```
#include "event.h"
```

in the files that want to use the libevent API. This also includes *ev.h*.

You need the following additional files for this:

```
event.h
event.c
```

AUTOCONF SUPPORT

Instead of using `EV_STANDALONE=1` and providing your configuration in whatever way you want, you can also `m4_include([libev.m4])` in your *configure.ac* and leave `EV_STANDALONE` undefined. *ev.c* will then include *config.h* and configure itself accordingly.

For this of course you need the m4 file:

```
libev.m4
```

PREPROCESSOR SYMBOLS/MACROS

Libev can be configured via a variety of preprocessor symbols you have to define before including (or compiling) any of its files. The default in the absence of autoconf is documented for every option.

Symbols marked with "(h)" do not change the ABI, and can have different values when compiling libev vs. including *ev.h*, so it is permissible to redefine them before including *ev.h* without breaking compatibility to a compiled library. All other symbols change the ABI, which means all users of libev and the libev code itself must be compiled with compatible settings.

EV_COMPAT3 (h)

Backwards compatibility is a major concern for libev. This is why this release of libev comes with wrappers for the functions and symbols that have been renamed between libev version 3 and 4.

You can disable these wrappers (to test compatibility with future versions) by defining `EV_COMPAT3` to `0` when compiling your sources. This has the additional advantage that you can drop the `struct` from `struct ev_loop` declarations, as libev will provide an `ev_loop` typedef in that case.

In some future version, the default for `EV_COMPAT3` will become `0`, and in some even more future version the compatibility code will be removed completely.

EV_STANDALONE (h)

Must always be `1` if you do not use autoconf configuration, which keeps libev from including *config.h*, and it also defines dummy implementations for some libevent functions (such as logging, which is not supported). It will also not define any of the structs usually found in *event.h* that are not directly supported by the libev core alone.

In standalone mode, libev will still try to automatically deduce the configuration, but has to be more conservative.

EV_USE_FLOOR

If defined to be `1`, libev will use the `floor ()` function for its periodic reschedule calculations, otherwise libev will fall back on a portable (slower) implementation. If you enable this, you usually have to link against libm or something equivalent. Enabling this when the `floor` function is not available will fail, so the safe default is to not enable this.

EV_USE_MONOTONIC

If defined to be `1`, libev will try to detect the availability of the monotonic clock option at both compile time and runtime. Otherwise no use of the monotonic clock option will be attempted. If you enable this, you usually have to link against librt or something similar. Enabling it when the functionality isn't available is safe, though, although you have to make sure you link against any libraries where the `clock_gettime` function is hiding in (often *-lrt*). See also `EV_USE_CLOCK_SYSCALL`.

EV_USE_REALTIME

If defined to be `1`, libev will try to detect the availability of the real-time clock option at compile time (and assume its availability at runtime if successful). Otherwise no use of the real-time clock option will be attempted. This effectively replaces `gettimeofday` by `clock_get (CLOCK_REALTIME, ...)` and will not normally affect correctness. See the note about libraries in the description of `EV_USE_MONOTONIC`, though. Defaults to the opposite value of

EV_USE_CLOCK_SYSCALL.

EV_USE_CLOCK_SYSCALL
: If defined to be 1, libev will try to use a direct syscall instead of calling the system-provided clock_gettime function. This option exists because on GNU/Linux, clock_gettime is in librt, but librt unconditionally pulls in libpthread, slowing down single-threaded programs needlessly. Using a direct syscall is slightly slower (in theory), because no optimised vdso implementation can be used, but avoids the pthread dependency. Defaults to 1 on GNU/Linux with glibc 2.x or higher, as it simplifies linking (no need for -lrt).

EV_USE_NANOSLEEP
: If defined to be 1, libev will assume that nanosleep () is available and will use it for delays. Otherwise it will use select ().

EV_USE_EVENTFD
: If defined to be 1, then libev will assume that eventfd () is available and will probe for kernel support at runtime. This will improve ev_signal and ev_async performance and reduce resource consumption. If undefined, it will be enabled if the headers indicate GNU/Linux + Glibc 2.7 or newer, otherwise disabled.

EV_USE_SELECT
: If undefined or defined to be 1, libev will compile in support for the select(2) backend. No attempt at auto-detection will be done: if no other method takes over, select will be it. Otherwise the select backend will not be compiled in.

EV_SELECT_USE_FD_SET
: If defined to 1, then the select backend will use the system fd_set structure. This is useful if libev doesn't compile due to a missing NFDBITS or fd_mask definition or it mis-guesses the bitset layout on exotic systems. This usually limits the range of file descriptors to some low limit such as 1024 or might have other limitations (winsocket only allows 64 sockets). The FD_SETSIZE macro, set before compilation, configures the maximum size of the fd_set.

EV_SELECT_IS_WINSOCKET
: When defined to 1, the select backend will assume that select/socket/connect etc. don't understand file descriptors but wants osf handles on win32 (this is the case when the select to be used is the winsock select). This means that it will call _get_osfhandle on the fd to convert it to an OS handle. Otherwise, it is assumed that all these functions actually work on fds, even on win32. Should not be defined on non-win32 platforms.

EV_FD_TO_WIN32_HANDLE(fd)
: If EV_SELECT_IS_WINSOCKET is enabled, then libev needs a way to map file descriptors to socket handles. When not defining this symbol (the default), then libev will call _get_osfhandle, which is usually correct. In some cases, programs use their own file descriptor management, in which case they can provide this function to map fds to socket handles.

EV_WIN32_HANDLE_TO_FD(handle)
: If EV_SELECT_IS_WINSOCKET then libev maps handles to file descriptors using the standard _open_osfhandle function. For programs implementing their own fd to handle mapping, overwriting this function makes it easier to do so. This can be done by defining this macro to an appropriate value.

EV_WIN32_CLOSE_FD(fd)
: If programs implement their own fd to handle mapping on win32, then this macro can be used to override the close function, useful to unregister file descriptors again. Note that the replacement function has to close the underlying OS handle.

EV_USE_WSASOCKET
: If defined to be 1, libev will use WSASocket to create its internal communication socket, which works better in some environments. Otherwise, the normal socket function will be used, which works better in other environments.

EV_USE_POLL
: If defined to be `1`, libev will compile in support for the `poll`(2) backend. Otherwise it will be enabled on non−win32 platforms. It takes precedence over select.

EV_USE_EPOLL
: If defined to be `1`, libev will compile in support for the Linux `epoll`(7) backend. Its availability will be detected at runtime, otherwise another method will be used as fallback. This is the preferred backend for GNU/Linux systems. If undefined, it will be enabled if the headers indicate GNU/Linux + Glibc 2.4 or newer, otherwise disabled.

EV_USE_LINUXAIO
: If defined to be `1`, libev will compile in support for the Linux aio backend. Due to it's currenbt limitations it has to be requested explicitly. If undefined, it will be enabled on linux, otherwise disabled.

EV_USE_KQUEUE
: If defined to be `1`, libev will compile in support for the BSD style `kqueue`(2) backend. Its actual availability will be detected at runtime, otherwise another method will be used as fallback. This is the preferred backend for BSD and BSD-like systems, although on most BSDs kqueue only supports some types of fds correctly (the only platform we found that supports ptys for example was NetBSD), so kqueue might be compiled in, but not be used unless explicitly requested. The best way to use it is to find out whether kqueue supports your type of fd properly and use an embedded kqueue loop.

EV_USE_PORT
: If defined to be `1`, libev will compile in support for the Solaris 10 port style backend. Its availability will be detected at runtime, otherwise another method will be used as fallback. This is the preferred backend for Solaris 10 systems.

EV_USE_DEVPOLL
: Reserved for future expansion, works like the USE symbols above.

EV_USE_INOTIFY
: If defined to be `1`, libev will compile in support for the Linux inotify interface to speed up `ev_stat` watchers. Its actual availability will be detected at runtime. If undefined, it will be enabled if the headers indicate GNU/Linux + Glibc 2.4 or newer, otherwise disabled.

EV_NO_SMP
: If defined to be `1`, libev will assume that memory is always coherent between threads, that is, threads can be used, but threads never run on different cpus (or different cpu cores). This reduces dependencies and makes libev faster.

EV_NO_THREADS
: If defined to be `1`, libev will assume that it will never be called from different threads (that includes signal handlers), which is a stronger assumption than `EV_NO_SMP`, above. This reduces dependencies and makes libev faster.

EV_ATOMIC_T
: Libev requires an integer type (suitable for storing `0` or `1`) whose access is atomic with respect to other threads or signal contexts. No such type is easily found in the C language, so you can provide your own type that you know is safe for your purposes. It is used both for signal handler "locking" as well as for signal and thread safety in `ev_async` watchers.

  In the absence of this define, libev will use `sig_atomic_t volatile` (from *signal.h*), which is usually good enough on most platforms.

EV_H (h)
: The name of the *ev.h* header file used to include it. The default if undefined is `"ev.h"` in *event.h*, *ev.c* and *ev++.h*. This can be used to virtually rename the *ev.h* header file in case of conflicts.

EV_CONFIG_H (h)
: If `EV_STANDALONE` isn't `1`, this variable can be used to override *ev.c*'s idea of where to find the *config.h* file, similarly to `EV_H`, above.

EV_EVENT_H (h)
: Similarly to `EV_H`, this macro can be used to override *event.c*'s idea of how the *event.h* header can be found, the default is `"event.h"`.

EV_PROTOTYPES (h)
: If defined to be `0`, then *ev.h* will not define any function prototypes, but still define all the structs and other symbols. This is occasionally useful if you want to provide your own wrapper functions around libev functions.

EV_MULTIPLICITY
: If undefined or defined to `1`, then all event-loop-specific functions will have the `struct ev_loop *` as first argument, and you can create additional independent event loops. Otherwise there will be no support for multiple event loops and there is no first event loop pointer argument. Instead, all functions act on the single default loop.

  Note that `EV_DEFAULT` and `EV_DEFAULT_` will no longer provide a default loop when multiplicity is switched off – you always have to initialise the loop manually in this case.

EV_MINPRI
EV_MAXPRI
: The range of allowed priorities. `EV_MINPRI` must be smaller or equal to `EV_MAXPRI`, but otherwise there are no non-obvious limitations. You can provide for more priorities by overriding those symbols (usually defined to be `-2` and `2`, respectively).

  When doing priority-based operations, libev usually has to linearly search all the priorities, so having many of them (hundreds) uses a lot of space and time, so using the defaults of five priorities (-2 .. +2) is usually fine.

  If your embedding application does not need any priorities, defining these both to `0` will save some memory and CPU.

EV_PERIODIC_ENABLE, EV_IDLE_ENABLE, EV_EMBED_ENABLE, EV_STAT_ENABLE, EV_PREPARE_ENABLE, EV_CHECK_ENABLE, EV_FORK_ENABLE, EV_SIGNAL_ENABLE, EV_ASYNC_ENABLE, EV_CHILD_ENABLE.
: If undefined or defined to be `1` (and the platform supports it), then the respective watcher type is supported. If defined to be `0`, then it is not. Disabling watcher types mainly saves code size.

EV_FEATURES
: If you need to shave off some kilobytes of code at the expense of some speed (but with the full API), you can define this symbol to request certain subsets of functionality. The default is to enable all features that can be enabled on the platform.

  A typical way to use this symbol is to define it to `0` (or to a bitset with some broad features you want) and then selectively re-enable additional parts you want, for example if you want everything minimal, but multiple event loop support, async and child watchers and the poll backend, use this:

```
#define EV_FEATURES 0
#define EV_MULTIPLICITY 1
#define EV_USE_POLL 1
#define EV_CHILD_ENABLE 1
#define EV_ASYNC_ENABLE 1
```

  The actual value is a bitset, it can be a combination of the following values (by default, all of these are enabled):

  `1` – faster/larger code
  : Use larger code to speed up some operations.

    Currently this is used to override some inlining decisions (enlarging the code size by roughly 30% on amd64).

    When optimising for size, use of compiler flags such as `-Os` with gcc is recommended, as well as

-DNDEBUG, as libev contains a number of assertions.

The default is off when __OPTIMIZE_SIZE__ is defined by your compiler (e.g. gcc with -Os).

2 – faster/larger data structures
Replaces the small 2–heap for timer management by a faster 4–heap, larger hash table sizes and so on. This will usually further increase code size and can additionally have an effect on the size of data structures at runtime.

The default is off when __OPTIMIZE_SIZE__ is defined by your compiler (e.g. gcc with -Os).

4 – full API configuration
This enables priorities (sets EV_MAXPRI=2 and EV_MINPRI=–2), and enables multiplicity (EV_MULTIPLICITY=1).

8 – full API
This enables a lot of the "lesser used" API functions. See ev.h for details on which parts of the API are still available without this feature, and do not complain if this subset changes over time.

16 – enable all optional watcher types
Enables all optional watcher types. If you want to selectively enable only some watcher types other than I/O and timers (e.g. prepare, embed, async, child...) you can enable them manually by defining EV_watchertype_ENABLE to 1 instead.

32 – enable all backends
This enables all backends – without this feature, you need to enable at least one backend manually (EV_USE_SELECT is a good choice).

64 – enable OS-specific "helper" APIs
Enable inotify, eventfd, signalfd and similar OS-specific helper APIs by default.

Compiling with gcc -Os -DEV_STANDALONE -DEV_USE_EPOLL=1 -DEV_FEATURES=0 reduces the compiled size of libev from 24.7Kb code/2.8Kb data to 6.5Kb code/0.3Kb data on my GNU/Linux amd64 system, while still giving you I/O watchers, timers and monotonic clock support.

With an intelligent-enough linker (gcc+binutils are intelligent enough when you use -Wl,--gc-sections -ffunction-sections) functions unused by your program might be left out as well – a binary starting a timer and an I/O watcher then might come out at only 5Kb.

EV_API_STATIC
If this symbol is defined (by default it is not), then all identifiers will have static linkage. This means that libev will not export any identifiers, and you cannot link against libev anymore. This can be useful when you embed libev, only want to use libev functions in a single file, and do not want its identifiers to be visible.

To use this, define EV_API_STATIC and include *ev.c* in the file that wants to use libev.

This option only works when libev is compiled with a C compiler, as C++ doesn't support the required declaration syntax.

EV_AVOID_STDIO
If this is set to 1 at compiletime, then libev will avoid using stdio functions (printf, scanf, perror etc.). This will increase the code size somewhat, but if your program doesn't otherwise depend on stdio and your libc allows it, this avoids linking in the stdio library which is quite big.

Note that error messages might become less precise when this option is enabled.

EV_NSIG
The highest supported signal number, +1 (or, the number of signals): Normally, libev tries to deduce the maximum number of signals automatically, but sometimes this fails, in which case it can be specified. Also, using a lower number than detected (32 should be good for about any system in

existence) can save some memory, as libev statically allocates some 12−24 bytes per signal number.

EV_PID_HASHSIZE
: `ev_child` watchers use a small hash table to distribute workload by pid. The default size is `16` (or `1` with `EV_FEATURES` disabled), usually more than enough. If you need to manage thousands of children you might want to increase this value (*must* be a power of two).

EV_INOTIFY_HASHSIZE
: `ev_stat` watchers use a small hash table to distribute workload by inotify watch id. The default size is `16` (or `1` with `EV_FEATURES` disabled), usually more than enough. If you need to manage thousands of `ev_stat` watchers you might want to increase this value (*must* be a power of two).

EV_USE_4HEAP
: Heaps are not very cache-efficient. To improve the cache-efficiency of the timer and periodics heaps, libev uses a 4−heap when this symbol is defined to `1`. The 4−heap uses more complicated (longer) code but has noticeably faster performance with many (thousands) of watchers.

  The default is `1`, unless `EV_FEATURES` overrides it, in which case it will be `0`.

EV_HEAP_CACHE_AT
: Heaps are not very cache-efficient. To improve the cache-efficiency of the timer and periodics heaps, libev can cache the timestamp (*at*) within the heap structure (selected by defining `EV_HEAP_CACHE_AT` to `1`), which uses 8−12 bytes more per watcher and a few hundred bytes more code, but avoids random read accesses on heap changes. This improves performance noticeably with many (hundreds) of watchers.

  The default is `1`, unless `EV_FEATURES` overrides it, in which case it will be `0`.

EV_VERIFY
: Controls how much internal verification (see `ev_verify ()`) will be done: If set to `0`, no internal verification code will be compiled in. If set to `1`, then verification code will be compiled in, but not called. If set to `2`, then the internal verification code will be called once per loop, which can slow down libev. If set to `3`, then the verification code will be called very frequently, which will slow down libev considerably.

  Verification errors are reported via C's `assert` mechanism, so if you disable that (e.g. by defining `NDEBUG`) then no errors will be reported.

  The default is `1`, unless `EV_FEATURES` overrides it, in which case it will be `0`.

EV_COMMON
: By default, all watchers have a `void *data` member. By redefining this macro to something else you can include more and other types of members. You have to define it each time you include one of the files, though, and it must be identical each time.

  For example, the perl EV module uses something like this:

```
#define EV_COMMON                       \
  SV *self; /* contains this struct */  \
  SV *cb_sv, *fh /* note no trailing ";" */
```

EV_CB_DECLARE (type)
EV_CB_INVOKE (watcher, revents)
ev_set_cb (ev, cb)
: Can be used to change the callback member declaration in each watcher, and the way callbacks are invoked and set. Must expand to a struct member definition and a statement, respectively. See the *ev.h* header file for their default definitions. One possible use for overriding these is to avoid the `struct ev_loop *` as first argument in all cases, or to use method calls instead of plain function calls in C++.

EXPORTED API SYMBOLS

If you need to re-export the API (e.g. via a DLL) and you need a list of exported symbols, you can use the provided *Symbol.** files which list all public symbols, one per line:

```
Symbols.ev      for libev proper
Symbols.event   for the libevent emulation
```

This can also be used to rename all public symbols to avoid clashes with multiple versions of libev linked together (which is obviously bad in itself, but sometimes it is inconvenient to avoid this).

A sed command like this will create wrapper `#define`'s that you need to include before including *ev.h*:

```
<Symbols.ev sed -e "s/.*/#define & myprefix_&/" >wrap.h
```

This would create a file *wrap.h* which essentially looks like this:

```
#define ev_backend     myprefix_ev_backend
#define ev_check_start myprefix_ev_check_start
#define ev_check_stop  myprefix_ev_check_stop
...
```

EXAMPLES

For a real-world example of a program the includes libev verbatim, you can have a look at the EV perl module (<http://software.schmorp.de/pkg/EV.html>). It has the libev files in the *libev/* subdirectory and includes them in the *EV/EVAPI.h* (public interface) and *EV.xs* (implementation) files. Only the *EV.xs* file will be compiled. It is pretty complex because it provides its own header file.

The usage in rxvt-unicode is simpler. It has a *ev_cpp.h* header file that everybody includes and which overrides some configure choices:

```
#define EV_FEATURES 8
#define EV_USE_SELECT 1
#define EV_PREPARE_ENABLE 1
#define EV_IDLE_ENABLE 1
#define EV_SIGNAL_ENABLE 1
#define EV_CHILD_ENABLE 1
#define EV_USE_STDEXCEPT 0
#define EV_CONFIG_H <config.h>

#include "ev++.h"
```

And a *ev_cpp.C* implementation file that contains libev proper and is compiled:

```
#include "ev_cpp.h"
#include "ev.c"
```

INTERACTION WITH OTHER PROGRAMS, LIBRARIES OR THE ENVIRONMENT

THREADS AND COROUTINES

THREADS

All libev functions are reentrant and thread-safe unless explicitly documented otherwise, but libev implements no locking itself. This means that you can use as many loops as you want in parallel, as long as there are no concurrent calls into any libev function with the same loop parameter (`ev_default_*` calls have an implicit default loop parameter, of course): libev guarantees that different event loops share no data structures that need any locking.

Or to put it differently: calls with different loop parameters can be done concurrently from multiple threads, calls with the same loop parameter must be done serially (but can be done from different threads, as long as only one thread ever is inside a call at any point in time, e.g. by using a mutex per loop).

Specifically to support threads (and signal handlers), libev implements so-called `ev_async` watchers, which allow some limited form of concurrency on the same event loop, namely waking it up "from the outside".

If you want to know which design (one loop, locking, or multiple loops without or something else still) is best for your problem, then I cannot help you, but here is some generic advice:

- most applications have a main thread: use the default libev loop in that thread, or create a separate thread running only the default loop.

  This helps integrating other libraries or software modules that use libev themselves and don't care/know about threading.

- one loop per thread is usually a good model.

  Doing this is almost never wrong, sometimes a better-performance model exists, but it is always a good start.

- other models exist, such as the leader/follower pattern, where one loop is handed through multiple threads in a kind of round-robin fashion.

  Choosing a model is hard – look around, learn, know that usually you can do better than you currently do :–)

- often you need to talk to some other thread which blocks in the event loop.

  `ev_async` watchers can be used to wake them up from other threads safely (or from signal contexts...).

  An example use would be to communicate signals or other events that only work in the default loop by registering the signal watcher with the default loop and triggering an `ev_async` watcher from the default loop watcher callback into the event loop interested in the signal.

See also ''THREAD LOCKING EXAMPLE''.

COROUTINES

Libev is very accommodating to coroutines (''cooperative threads''): libev fully supports nesting calls to its functions from different coroutines (e.g. you can call `ev_run` on the same loop from two different coroutines, and switch freely between both coroutines running the loop, as long as you don't confuse yourself). The only exception is that you must not do this from `ev_periodic` reschedule callbacks.

Care has been taken to ensure that libev does not keep local state inside `ev_run`, and other calls do not usually allow for coroutine switches as they do not call any callbacks.

COMPILER WARNINGS

Depending on your compiler and compiler settings, you might get no or a lot of warnings when compiling libev code. Some people are apparently scared by this.

However, these are unavoidable for many reasons. For one, each compiler has different warnings, and each user has different tastes regarding warning options. ''Warn-free'' code therefore cannot be a goal except when targeting a specific compiler and compiler-version.

Another reason is that some compiler warnings require elaborate workarounds, or other changes to the code that make it less clear and less maintainable.

And of course, some compiler warnings are just plain stupid, or simply wrong (because they don't actually warn about the condition their message seems to warn about). For example, certain older gcc versions had some warnings that resulted in an extreme number of false positives. These have been fixed, but some people still insist on making code warn-free with such buggy versions.

While libev is written to generate as few warnings as possible, ''warn-free'' code is not a goal, and it is recommended not to build libev with any compiler warnings enabled unless you are prepared to cope with them (e.g. by ignoring them). Remember that warnings are just that: warnings, not errors, or proof of bugs.

VALGRIND

Valgrind has a special section here because it is a popular tool that is highly useful. Unfortunately, valgrind reports are very hard to interpret.

If you think you found a bug (memory leak, uninitialised data access etc.) in libev, then check twice: If valgrind reports something like:

```
==2274==    definitely lost: 0 bytes in 0 blocks.
==2274==      possibly lost: 0 bytes in 0 blocks.
==2274==    still reachable: 256 bytes in 1 blocks.
```

Then there is no memory leak, just as memory accounted to global variables is not a memleak – the memory is still being referenced, and didn't leak.

Similarly, under some circumstances, valgrind might report kernel bugs as if it were a bug in libev (e.g. in realloc or in the poll backend, although an acceptable workaround has been found here), or it might be confused.

Keep in mind that valgrind is a very good tool, but only a tool. Don't make it into some kind of religion.

If you are unsure about something, feel free to contact the mailing list with the full valgrind report and an explanation on why you think this is a bug in libev (best check the archives, too :). However, don't be annoyed when you get a brisk "this is no bug" answer and take the chance of learning how to interpret valgrind properly.

If you need, for some reason, empty reports from valgrind for your project I suggest using suppression lists.

PORTABILITY NOTES

GNU/LINUX 32 BIT LIMITATIONS

GNU/Linux is the only common platform that supports 64 bit file/large file interfaces but *disables* them by default.

That means that libev compiled in the default environment doesn't support files larger than 2GiB or so, which mainly affects `ev_stat` watchers.

Unfortunately, many programs try to work around this GNU/Linux issue by enabling the large file API, which makes them incompatible with the standard libev compiled for their system.

Likewise, libev cannot enable the large file API itself as this would suddenly make it incompatible to the default compile time environment, i.e. all programs not using special compile switches.

OS/X AND DARWIN BUGS

The whole thing is a bug if you ask me – basically any system interface you touch is broken, whether it is locales, poll, kqueue or even the OpenGL drivers.

`kqueue` is buggy

The kqueue syscall is broken in all known versions – most versions support only sockets, many support pipes.

Libev tries to work around this by not using `kqueue` by default on this rotten platform, but of course you can still ask for it when creating a loop – embedding a socket-only kqueue loop into a select-based one is probably going to work well.

`poll` is buggy

Instead of fixing `kqueue`, Apple replaced their (working) `poll` implementation by something calling `kqueue` internally around the 10.5.6 release, so now `kqueue` *and* `poll` are broken.

Libev tries to work around this by not using `poll` by default on this rotten platform, but of course you can still ask for it when creating a loop.

`select` is buggy

All that's left is `select`, and of course Apple found a way to fuck this one up as well: On OS/X, `select` actively limits the number of file descriptors you can pass in to 1024 – your program suddenly crashes when you use more.

There is an undocumented "workaround" for this – defining `_DARWIN_UNLIMITED_SELECT`, which libev tries to use, so select *should* work on OS/X.

SOLARIS PROBLEMS AND WORKAROUNDS

errno reentrancy

The default compile environment on Solaris is unfortunately so thread-unsafe that you can't even use components/libraries compiled without `-D_REENTRANT` in a threaded program, which, of course, isn't defined by default. A valid, if stupid, implementation choice.

If you want to use libev in threaded environments you have to make sure it's compiled with `_REENTRANT` defined.

Event port backend

The scalable event interface for Solaris is called "event ports". Unfortunately, this mechanism is very buggy in all major releases. If you run into high CPU usage, your program freezes or you get a large number of spurious wakeups, make sure you have all the relevant and latest kernel patches applied. No, I don't know which ones, but there are multiple ones to apply, and afterwards, event ports actually work great.

If you can't get it to work, you can try running the program by setting the environment variable `LIBEV_FLAGS=3` to only allow `poll` and `select` backends.

AIX POLL BUG

AIX unfortunately has a broken `poll.h` header. Libev works around this by trying to avoid the poll backend altogether (i.e. it's not even compiled in), which normally isn't a big problem as `select` works fine with large bitsets on AIX, and AIX is dead anyway.

WIN32 PLATFORM LIMITATIONS AND WORKAROUNDS

General issues

Win32 doesn't support any of the standards (e.g. POSIX) that libev requires, and its I/O model is fundamentally incompatible with the POSIX model. Libev still offers limited functionality on this platform in the form of the `EVBACKEND_SELECT` backend, and only supports socket descriptors. This only applies when using Win32 natively, not when using e.g. cygwin. Actually, it only applies to the microsofts own compilers, as every compiler comes with a slightly differently broken/incompatible environment.

Lifting these limitations would basically require the full re-implementation of the I/O system. If you are into this kind of thing, then note that glib does exactly that for you in a very portable way (note also that glib is the slowest event library known to man).

There is no supported compilation method available on windows except embedding it into other applications.

Sensible signal handling is officially unsupported by Microsoft – libev tries its best, but under most conditions, signals will simply not work.

Not a libev limitation but worth mentioning: windows apparently doesn't accept large writes: instead of resulting in a partial write, windows will either accept everything or return `ENOBUFS` if the buffer is too large, so make sure you only write small amounts into your sockets (less than a megabyte seems safe, but this apparently depends on the amount of memory available).

Due to the many, low, and arbitrary limits on the win32 platform and the abysmal performance of winsockets, using a large number of sockets is not recommended (and not reasonable). If your program needs to use more than a hundred or so sockets, then likely it needs to use a totally different implementation for windows, as libev offers the POSIX readiness notification model, which cannot be implemented efficiently on windows (due to Microsoft monopoly games).

A typical way to use libev under windows is to embed it (see the embedding section for details) and use the following *evwrap.h* header file instead of *ev.h*:

```
#define EV_STANDALONE              /* keeps ev from requiring config.h */
#define EV_SELECT_IS_WINSOCKET 1   /* configure libev for windows select */

#include "ev.h"
```

And compile the following *evwrap.c* file into your project (make sure you do *not* compile the *ev.c* or any other embedded source files!):

```
#include "evwrap.h"
#include "ev.c"
```

The winsocket `select` function

The winsocket `select` function doesn't follow POSIX in that it requires socket *handles* and not socket *file descriptors* (it is also extremely buggy). This makes select very inefficient, and also requires a mapping from file descriptors to socket handles (the Microsoft C runtime provides the function `_open_osfhandle` for this). See the discussion of the `EV_SELECT_USE_FD_SET`, `EV_SELECT_IS_WINSOCKET` and `EV_FD_TO_WIN32_HANDLE` preprocessor symbols for more info.

The configuration for a "naked" win32 using the Microsoft runtime libraries and raw winsocket select is:

```
#define EV_USE_SELECT 1
#define EV_SELECT_IS_WINSOCKET 1   /* forces EV_SELECT_USE_FD_SET, too */
```

Note that winsockets handling of fd sets is O(n), so you can easily get a complexity in the O(n²) range when using win32.

Limited number of file descriptors

Windows has numerous arbitrary (and low) limits on things.

Early versions of winsocket's select only supported waiting for a maximum of `64` handles (probably owning to the fact that all windows kernels can only wait for `64` things at the same time internally; Microsoft recommends spawning a chain of threads and wait for 63 handles and the previous thread in each. Sounds great!).

Newer versions support more handles, but you need to define `FD_SETSIZE` to some high number (e.g. `2048`) before compiling the winsocket select call (which might be in libev or elsewhere, for example, perl and many other interpreters do their own select emulation on windows).

Another limit is the number of file descriptors in the Microsoft runtime libraries, which by default is `64` (there must be a hidden *64* fetish or something like this inside Microsoft). You can increase this by calling `_setmaxstdio`, which can increase this limit to `2048` (another arbitrary limit), but is broken in many versions of the Microsoft runtime libraries. This might get you to about `512` or `2048` sockets (depending on windows version and/or the phase of the moon). To get more, you need to wrap all I/O functions and provide your own fd management, but the cost of calling select (O(n²)) will likely make this unworkable.

PORTABILITY REQUIREMENTS

In addition to a working ISO-C implementation and of course the backend-specific APIs, libev relies on a few additional extensions:

`void (*)(ev_watcher_type *, int revents)` must have compatible calling conventions regardless of `ev_watcher_type *`.

Libev assumes not only that all watcher pointers have the same internal structure (guaranteed by POSIX but not by ISO C for example), but it also assumes that the same (machine) code can be used to call any watcher callback: The watcher callbacks have different type signatures, but libev calls them using an `ev_watcher *` internally.

null pointers and integer zero are represented by 0 bytes

Libev uses `memset` to initialise structs and arrays to `0` bytes, and relies on this setting pointers and integers to null.

pointer accesses must be thread-atomic

Accessing a pointer value must be atomic, it must both be readable and writable in one piece – this is the case on all current architectures.

`sig_atomic_t volatile` must be thread-atomic as well

The type `sig_atomic_t volatile` (or whatever is defined as `EV_ATOMIC_T`) must be atomic with respect to accesses from different threads. This is not part of the specification for

`sig_atomic_t`, but is believed to be sufficiently portable.

`sigprocmask` must work in a threaded environment
: Libev uses `sigprocmask` to temporarily block signals. This is not allowed in a threaded program (`pthread_sigmask` has to be used). Typical pthread implementations will either allow `sigprocmask` in the "main thread" or will block signals process-wide, both behaviours would be compatible with libev. Interaction between `sigprocmask` and `pthread_sigmask` could complicate things, however.

  The most portable way to handle signals is to block signals in all threads except the initial one, and run the signal handling loop in the initial thread as well.

`long` must be large enough for common memory allocation sizes
: To improve portability and simplify its API, libev uses `long` internally instead of `size_t` when allocating its data structures. On non-POSIX systems (Microsoft...) this might be unexpectedly low, but is still at least 31 bits everywhere, which is enough for hundreds of millions of watchers.

`double` must hold a time value in seconds with enough accuracy
: The type `double` is used to represent timestamps. It is required to have at least 51 bits of mantissa (and 9 bits of exponent), which is good enough for at least into the year 4000 with millisecond accuracy (the design goal for libev). This requirement is overfulfilled by implementations using IEEE 754, which is basically all existing ones.

  With IEEE 754 doubles, you get microsecond accuracy until at least the year 2255 (and millisecond accuracy till the year 287396 – by then, libev is either obsolete or somebody patched it to use `long double` or something like that, just kidding).

If you know of other additional requirements drop me a note.

ALGORITHMIC COMPLEXITIES

In this section the complexities of (many of) the algorithms used inside libev will be documented. For complexity discussions about backends see the documentation for `ev_default_init`.

All of the following are about amortised time: If an array needs to be extended, libev needs to realloc and move the whole array, but this happens asymptotically rarer with higher number of elements, so O(1) might mean that libev does a lengthy realloc operation in rare cases, but on average it is much faster and asymptotically approaches constant time.

Starting and stopping timer/periodic watchers: O(log skipped_other_timers)
: This means that, when you have a watcher that triggers in one hour and there are 100 watchers that would trigger before that, then inserting will have to skip roughly seven (`ld 100`) of these watchers.

Changing timer/periodic watchers (by autorepeat or calling again): O(log skipped_other_timers)
: That means that changing a timer costs less than removing/adding them, as only the relative motion in the event queue has to be paid for.

Starting io/check/prepare/idle/signal/child/fork/async watchers: O(1)
: These just add the watcher into an array or at the head of a list.

Stopping check/prepare/idle/fork/async watchers: O(1)

Stopping an io/signal/child watcher: O(number_of_watchers_for_this_(fd/signal/pid % EV_PID_HASHSIZE))
: These watchers are stored in lists, so they need to be walked to find the correct watcher to remove. The lists are usually short (you don't usually have many watchers waiting for the same fd or signal: one is typical, two is rare).

Finding the next timer in each loop iteration: O(1)
: By virtue of using a binary or 4−heap, the next timer is always found at a fixed position in the storage array.

Each change on a file descriptor per loop iteration: O(number_of_watchers_for_this_fd)
: A change means an I/O watcher gets started or stopped, which requires libev to recalculate its status (and possibly tell the kernel, depending on backend and whether `ev_io_set` was used).

Activating one watcher (putting it into the pending state): O(1)

Priority handling: O(number_of_priorities)

Priorities are implemented by allocating some space for each priority. When doing priority-based operations, libev usually has to linearly search all the priorities, but starting/stopping and activating watchers becomes O(1) with respect to priority handling.

Sending an ev_async: O(1)

Processing ev_async_send: O(number_of_async_watchers)

Processing signals: O(max_signal_number)

Sending involves a system call *iff* there were no other `ev_async_send` calls in the current loop iteration and the loop is currently blocked. Checking for async and signal events involves iterating over all running async watchers or all signal numbers.

PORTING FROM LIBEV 3.X TO 4.X

The major version 4 introduced some incompatible changes to the API.

At the moment, the `ev.h` header file provides compatibility definitions for all changes, so most programs should still compile. The compatibility layer might be removed in later versions of libev, so better update to the new API early than late.

`EV_COMPAT3` backwards compatibility mechanism

The backward compatibility mechanism can be controlled by `EV_COMPAT3`. See "PREPROCESSOR SYMBOLS/MACROS" in the "EMBEDDING" section.

`ev_default_destroy` and `ev_default_fork` have been removed

These calls can be replaced easily by their `ev_loop_xxx` counterparts:

```
ev_loop_destroy (EV_DEFAULT_UC);
ev_loop_fork (EV_DEFAULT);
```

function/symbol renames

A number of functions and symbols have been renamed:

```
ev_loop          => ev_run
EVLOOP_NONBLOCK => EVRUN_NOWAIT
EVLOOP_ONESHOT  => EVRUN_ONCE

ev_unloop        => ev_break
EVUNLOOP_CANCEL => EVBREAK_CANCEL
EVUNLOOP_ONE     => EVBREAK_ONE
EVUNLOOP_ALL     => EVBREAK_ALL

EV_TIMEOUT       => EV_TIMER

ev_loop_count    => ev_iteration
ev_loop_depth    => ev_depth
ev_loop_verify   => ev_verify
```

Most functions working on `struct ev_loop` objects don't have an `ev_loop_` prefix, so it was removed; `ev_loop`, `ev_unloop` and associated constants have been renamed to not collide with the `struct ev_loop` anymore and `EV_TIMER` now follows the same naming scheme as all other watcher types. Note that `ev_loop_fork` is still called `ev_loop_fork` because it would otherwise clash with the `ev_fork` typedef.

`EV_MINIMAL` mechanism replaced by `EV_FEATURES`

The preprocessor symbol `EV_MINIMAL` has been replaced by a different mechanism, `EV_FEATURES`. Programs using `EV_MINIMAL` usually compile and work, but the library code will of course be larger.

GLOSSARY

active
: A watcher is active as long as it has been started and not yet stopped. See "WATCHER STATES" for details.

application
: In this document, an application is whatever is using libev.

backend
: The part of the code dealing with the operating system interfaces.

callback
: The address of a function that is called when some event has been detected. Callbacks are being passed the event loop, the watcher that received the event, and the actual event bitset.

callback/watcher invocation
: The act of calling the callback associated with a watcher.

event
: A change of state of some external event, such as data now being available for reading on a file descriptor, time having passed or simply not having any other events happening anymore.

  In libev, events are represented as single bits (such as `EV_READ` or `EV_TIMER`).

event library
: A software package implementing an event model and loop.

event loop
: An entity that handles and processes external events and converts them into callback invocations.

event model
: The model used to describe how an event loop handles and processes watchers and events.

pending
: A watcher is pending as soon as the corresponding event has been detected. See "WATCHER STATES" for details.

real time
: The physical time that is observed. It is apparently strictly monotonic :)

wall-clock time
: The time and date as shown on clocks. Unlike real time, it can actually be wrong and jump forwards and backwards, e.g. when you adjust your clock.

watcher
: A data structure that describes interest in certain events. Watchers need to be started (attached to an event loop) before they can receive events.

AUTHOR

Marc Lehmann <libev@schmorp.de>, with repeated corrections by Mikael Magnusson and Emanuele Giaquinta, and minor corrections by many others.