libuv API documentation

Release 1.9.0

libuv contributors

May 09, 2016

Contents

CHAPTER 1

Overview

libuv is a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by Node.js, but it's also used by Luvit, Julia, pyuv, and others.

Note: In case you find errors in this documentation you can help by sending pull requests!

CHAPTER 2

Features

- Full-featured event loop backed by epoll, kqueue, IOCP, event ports.
- Asynchronous TCP and UDP sockets
- Asynchronous DNS resolution
- Asynchronous file and file system operations
- File system events
- ANSI escape code controlled TTY
- IPC with socket sharing, using Unix domain sockets or named pipes (Windows)
- Child processes
- Thread pool
- Signal handling
- High resolution clock
- Threading and synchronization primitives

CHAPTER 3

Downloads

libuv can be downloaded from here.

CHAPTER 4

Installation

Installation instructions can be found on the README.

CHAPTER 5

Upgrading

Migration guides for different libuv versions, starting with 1.0.

5.1 libuv 0.10 -> 1.0.0 migration guide

Some APIs changed quite a bit throughout the 1.0.0 development process. Here is a migration guide for the most significant changes that happened after 0.10 was released.

5.1.1 Loop initialization and closing

In libuv 0.10 (and previous versions), loops were created with *uv_loop_new*, which allocated memory for a new loop and initialized it; and destroyed with *uv_loop_delete*, which destroyed the loop and freed the memory. Starting with 1.0, those are deprecated and the user is responsible for allocating the memory and then initializing the loop.

libuv 0.10

```
uv_loop_t* loop = uv_loop_new();
...
uv_loop_delete(loop);
```

libuv 1.0

```
uv_loop_t* loop = malloc(sizeof *loop);
uv_loop_init(loop);
...
uv_loop_close(loop);
free(loop);
```

Note: Error handling was omitted for brevity. Check the documentation for `uv_loop_init()` and `uv_loop_close()`.

5.1.2 Error handling

Error handling had a major overhaul in libuv 1.0. In general, functions and status parameters would get 0 for success and -1 for failure on libuv 0.10, and the user had to use *uv_last_error* to fetch the error code, which was a positive number.

In 1.0, functions and status parameters contain the actual error code, which is 0 for success, or a negative number in case of error.

libuv 0.10

```
... assume 'server' is a TCP server which is already listening
r = uv_listen((uv_stream_t*) server, 511, NULL);
if (r == -1) {
  uv_err_t err = uv_last_error(uv_default_loop());
  /* err.code contains UV_EADDRINUSE */
}
```

libuv 1.0

```
... assume 'server' is a TCP server which is already listening
r = uv_listen((uv_stream_t*) server, 511, NULL);
if (r < 0) {
  /* r contains UV_EADDRINUSE */
}
```

5.1.3 Threadpool changes

In libuv 0.10 Unix used a threadpool which defaulted to 4 threads, while Windows used the *QueueUserWorkItem* API, which uses a Windows internal threadpool, which defaults to 512 threads per process.

In 1.0, we unified both implementations, so Windows now uses the same implementation Unix does. The threadpool size can be set by exporting the `UV_THREADPOOL_SIZE` environment variable. See *Thread pool work scheduling*.

5.1.4 Allocation callback API change

In libuv 0.10 the callback had to return a filled `uv_buf_t` by value:

```
uv_buf_t alloc_cb(uv_handle_t* handle, size_t size) {
    return uv_buf_init(malloc(size), size);
}
```

In libuv 1.0 a pointer to a buffer is passed to the callback, which the user needs to fill:

```
void alloc_cb(uv_handle_t* handle, size_t size, uv_buf_t* buf) {
    buf->base = malloc(size);
    buf->len = size;
}
```

5.1.5 Unification of IPv4 / IPv6 APIs

libuv 1.0 unified the IPv4 and IPv6 APIS. There is no longer a *uv_tcp_bind* and *uv_tcp_bind6* duality, there is only `uv_tcp_bind()` now.

IPv4 functions took `struct sockaddr_in` structures by value, and IPv6 functions took `struct sockaddr_in6`. Now functions take a `struct sockaddr*` (note it's a pointer). It can be stack allocated.

libuv 0.10

```
struct sockaddr_in addr = uv_ip4_addr("0.0.0.0", 1234);
...
uv_tcp_bind(&server, addr)
```

libuv 1.0

```
struct sockaddr_in addr;
uv_ip4_addr("0.0.0.0", 1234, &addr)
...
uv_tcp_bind(&server, (const struct sockaddr*) &addr, 0);
```

The IPv4 and IPv6 struct creating functions (`uv_ip4_addr()` and `uv_ip6_addr()`) have also changed, make sure you check the documentation.

..note:: This change applies to all functions that made a distinction between IPv4 and IPv6 addresses.

5.1.6 Streams / UDP data receive callback API change

The streams and UDP data receive callbacks now get a pointer to a `uv_buf_t` buffer, not a structure by value.

libuv 0.10

```
void on_read(uv_stream_t* handle,
             ssize_t nread,
             uv_buf_t buf) {
    ...
}

void recv_cb(uv_udp_t* handle,
             ssize_t nread,
             uv_buf_t buf,
             struct sockaddr* addr,
             unsigned flags) {
    ...
}
```

libuv 1.0

```
void on_read(uv_stream_t* handle,
             ssize_t nread,
             const uv_buf_t* buf) {
    ...
}

void recv_cb(uv_udp_t* handle,
             ssize_t nread,
             const uv_buf_t* buf,
             const struct sockaddr* addr,
             unsigned flags) {
    ...
}
```

5.1.7 Receiving handles over pipes API change

In libuv 0.10 (and earlier versions) the *uv_read2_start* function was used to start reading data on a pipe, which could also result in the reception of handles over it. The callback for such function looked like this:

```
void on_read(uv_pipe_t* pipe,
             ssize_t nread,
             uv_buf_t buf,
             uv_handle_type pending) {
```

```
    ...
}
```

In libuv 1.0, *uv_read2_start* was removed, and the user needs to check if there are pending handles using `uv_pipe_pending_count()` and `uv_pipe_pending_type()` while in the read callback:

```
void on_read(uv_stream_t* handle,
             ssize_t nread,
             const uv_buf_t* buf) {
    ...
    while (uv_pipe_pending_count((uv_pipe_t*) handle) != 0) {
        pending = uv_pipe_pending_type((uv_pipe_t*) handle);
        ...
    }
    ...
}
```

5.1.8 Extracting the file descriptor out of a handle

While it wasn't supported by the API, users often accessed the libuv internals in order to get access to the file descriptor of a TCP handle, for example.

```
fd = handle->io_watcher.fd;
```

This is now properly exposed through the `uv_fileno()` function.

5.1.9 uv_fs_readdir rename and API change

uv_fs_readdir returned a list of strings in the *req->ptr* field upon completion in libuv 0.10. In 1.0, this function got renamed to `uv_fs_scandir()`, since it's actually implemented using `scandir(3)`.

In addition, instead of allocating a full list strings, the user is able to get one result at a time by using the `uv_fs_scandir_next()` function. This function does not need to make a roundtrip to the threadpool, because libuv will keep the list of *dents* returned by `scandir(3)` around.

CHAPTER 6

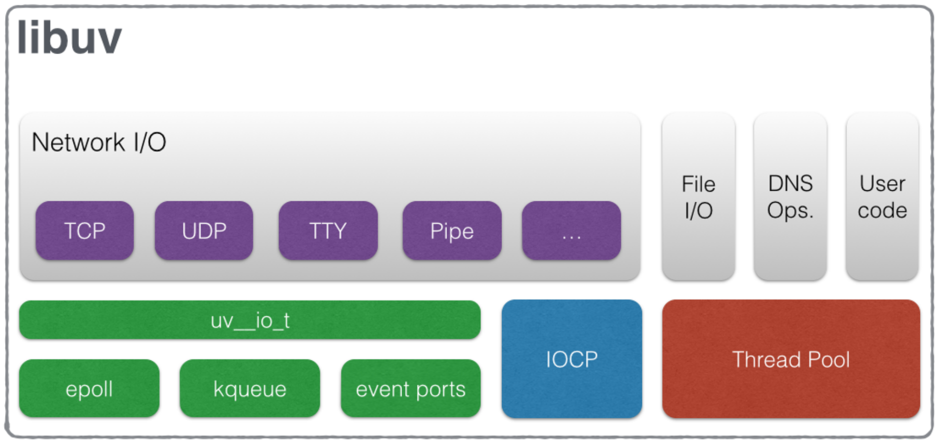
Documentation

6.1 Design overview

libuv is cross-platform support library which was originally written for NodeJS. It's designed around the event-driven asynchronous I/O model.

The library provides much more than simply abstraction over different I/O polling mechanisms: 'handles' and 'streams' provide a high level abstraction for sockets and other entities; cross-platform file I/O and threading functionality is also provided, amongst other things.

Here is a diagram illustrating the different parts that compose libuv and what subsystem they relate to:



6.1.1 Handles and requests

libuv provides users with 2 abstractions to work with, in combination with the event loop: handles and requests.

Handles represent long-lived objects capable of performing certain operations while active. Some examples: a prepare handle gets its callback called once every loop iteration when active, and a TCP server handle get its connection callback called every time there is a new connection.

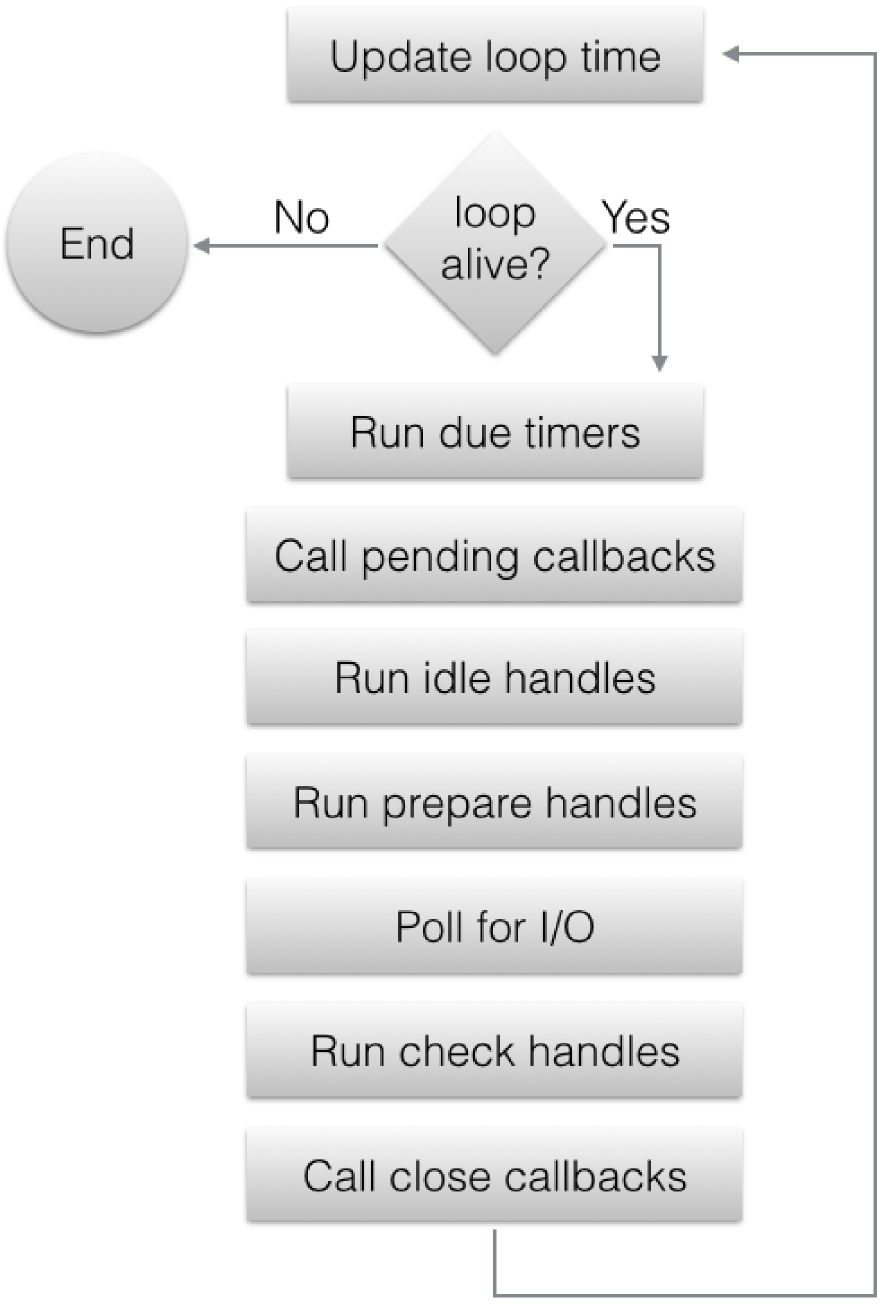
Requests represent (typically) short-lived operations. These operations can be performed over a handle: write requests are used to write data on a handle; or standalone: getaddrinfo requests don't need a handle they run directly on the loop.

6.1.2 The I/O loop

The I/O (or event) loop is the central part of libuv. It establishes the content for all I/O operations, and it's meant to be tied to a single thread. One can run multiple event loops as long as each runs in a different thread. The libuv event loop (or any other API involving the loop or handles, for that matter) **is not thread-safe** except where stated otherwise.

The event loop follows the rather usual single threaded asynchronous I/O approach: all (network) I/O is performed on non-blocking sockets which are polled using the best mechanism available on the given platform: epoll on Linux, kqueue on OSX and other BSDs, event ports on SunOS and IOCP on Windows. As part of a loop iteration the loop will block waiting for I/O activity on sockets which have been added to the poller and callbacks will be fired indicating socket conditions (readable, writable hangup) so handles can read, write or perform the desired I/O operation.

In order to better understand how the event loop operates, the following diagram illustrates all stages of a loop iteration:



1. The loop concept of 'now' is updated. The event loop caches the current time at the start of the event loop tick in order to reduce the number of time-related system calls.
2. If the loop is *alive* an iteration is started, otherwise the loop will exit immediately. So, when is a loop considered to be *alive*? If a loop has active and ref'd handles, active requests or closing handles it's considered to be *alive*.
3. Due timers are run. All active timers scheduled for a time before the loop's concept of *now* get their callbacks called.
4. Pending callbacks are called. All I/O callbacks are called right after polling for I/O, for the most part. There are cases, however, in which calling such a callback is deferred for the next loop iteration. If the previous iteration deferred any I/O callback it will be run at this point.
5. Idle handle callbacks are called. Despite the unfortunate name, idle handles are run on every loop iteration, if

they are active.

6. Prepare handle callbacks are called. Prepare handles get their callbacks called right before the loop will block for I/O.
7. Poll timeout is calculated. Before blocking for I/O the loop calculates for how long it should block. These are the rules when calculating the timeout:
   - If the loop was run with the `UV_RUN_NOWAIT` flag, the timeout is 0.
   - If the loop is going to be stopped (`uv_stop()` was called), the timeout is 0.
   - If there are no active handles or requests, the timeout is 0.
   - If there are any idle handles active, the timeout is 0.
   - If there are any handles pending to be closed, the timeout is 0.
   - If none of the above cases was matched, the timeout of the closest timer is taken, or if there are no active timers, infinity.
8. The loop blocks for I/O. At this point the loop will block for I/O for the timeout calculated on the previous step. All I/O related handles that were monitoring a given file descriptor for a read or write operation get their callbacks called at this point.
9. Check handle callbacks are called. Check handles get their callbacks called right after the loop has blocked for I/O. Check handles are essentially the counterpart of prepare handles.
10. Close callbacks are called. If a handle was closed by calling `uv_close()` it will get the close callback called.
11. Special case in case the loop was run with `UV_RUN_ONCE`, as it implies forward progress. It's possible that no I/O callbacks were fired after blocking for I/O, but some time has passed so there might be timers which are due, those timers get their callbacks called.
12. Iteration ends. If the loop was run with `UV_RUN_NOWAIT` or `UV_RUN_ONCE` modes the iteration is ended and `uv_run()` will return. If the loop was run with `UV_RUN_DEFAULT` it will continue from the start if it's still *alive*, otherwise it will also end.

Important: libuv uses a thread pool to make asynchronous file I/O operations possible, but network I/O is **always** performed in a single thread, each loop's thread.

Note: While the polling mechanism is different, libuv makes the execution model consistent across Unix systems and Windows.

6.1.3 File I/O

Unlike network I/O, there are no platform-specific file I/O primitives libuv could rely on, so the current approach is to run blocking file I/O operations in a thread pool.

For a thorough explanation of the cross-platform file I/O landscape, checkout this post.

libuv currently uses a global thread pool on which all loops can queue work on. 3 types of operations are currently run on this pool:

- Filesystem operations
- DNS functions (getaddrinfo and getnameinfo)
- User specified code via `uv_queue_work()`

Warning: See the *Thread pool work scheduling* section for more details, but keep in mind the thread pool size is quite limited.

6.2 Error handling

In libuv errors are negative numbered constants. As a rule of thumb, whenever there is a status parameter, or an API functions returns an integer, a negative number will imply an error.

Note: Implementation detail: on Unix error codes are the negated *errno* (or *-errno*), while on Windows they are defined by libuv to arbitrary negative numbers.

6.2.1 Error constants

`UV_E2BIG`
argument list too long

`UV_EACCES`
permission denied

`UV_EADDRINUSE`
address already in use

`UV_EADDRNOTAVAIL`
address not available

`UV_EAFNOSUPPORT`
address family not supported

`UV_EAGAIN`
resource temporarily unavailable

`UV_EAI_ADDRFAMILY`
address family not supported

`UV_EAI_AGAIN`
temporary failure

`UV_EAI_BADFLAGS`
bad ai_flags value

`UV_EAI_BADHINTS`
invalid value for hints

`UV_EAI_CANCELED`
request canceled

`UV_EAI_FAIL`
permanent failure

`UV_EAI_FAMILY`
ai_family not supported

`UV_EAI_MEMORY`
out of memory

UV_EAI_NODATA
no address

UV_EAI_NONAME
unknown node or service

UV_EAI_OVERFLOW
argument buffer overflow

UV_EAI_PROTOCOL
resolved protocol is unknown

UV_EAI_SERVICE
service not available for socket type

UV_EAI_SOCKTYPE
socket type not supported

UV_EALREADY
connection already in progress

UV_EBADF
bad file descriptor

UV_EBUSY
resource busy or locked

UV_ECANCELED
operation canceled

UV_ECHARSET
invalid Unicode character

UV_ECONNABORTED
software caused connection abort

UV_ECONNREFUSED
connection refused

UV_ECONNRESET
connection reset by peer

UV_EDESTADDRREQ
destination address required

UV_EEXIST
file already exists

UV_EFAULT
bad address in system call argument

UV_EFBIG
file too large

UV_EHOSTUNREACH
host is unreachable

UV_EINTR
interrupted system call

UV_EINVAL
invalid argument

UV_EIO
i/o error

UV_EISCONN
socket is already connected

UV_EISDIR
illegal operation on a directory

UV_ELOOP
too many symbolic links encountered

UV_EMFILE
too many open files

UV_EMSGSIZE
message too long

UV_ENAMETOOLONG
name too long

UV_ENETDOWN
network is down

UV_ENETUNREACH
network is unreachable

UV_ENFILE
file table overflow

UV_ENOBUFS
no buffer space available

UV_ENODEV
no such device

UV_ENOENT
no such file or directory

UV_ENOMEM
not enough memory

UV_ENONET
machine is not on the network

UV_ENOPROTOOPT
protocol not available

UV_ENOSPC
no space left on device

UV_ENOSYS
function not implemented

UV_ENOTCONN
socket is not connected

UV_ENOTDIR
not a directory

UV_ENOTEMPTY
directory not empty

UV_ENOTSOCK
socket operation on non-socket

UV_ENOTSUP
operation not supported on socket

UV_EPERM
operation not permitted

UV_EPIPE
broken pipe

UV_EPROTO
protocol error

UV_EPROTONOSUPPORT
protocol not supported

UV_EPROTOTYPE
protocol wrong type for socket

UV_ERANGE
result too large

UV_EROFS
read-only file system

UV_ESHUTDOWN
cannot send after transport endpoint shutdown

UV_ESPIPE
invalid seek

UV_ESRCH
no such process

UV_ETIMEDOUT
connection timed out

UV_ETXTBSY
text file is busy

UV_EXDEV
cross-device link not permitted

UV_UNKNOWN
unknown error

UV_EOF
end of file

UV_ENXIO
no such device or address

UV_EMLINK
too many links

6.2.2 API

const char* **uv_strerror**(int *err*)
Returns the error message for the given error code. Leaks a few bytes of memory when you call it with an unknown error code.

const char* **uv_err_name** (int *err*)
Returns the error name for the given error code. Leaks a few bytes of memory when you call it with an unknown error code.

6.3 Version-checking macros and functions

Starting with version 1.0.0 libuv follows the semantic versioning scheme. This means that new APIs can be introduced throughout the lifetime of a major release. In this section you'll find all macros and functions that will allow you to write or compile code conditionally, in order to work with multiple libuv versions.

6.3.1 Macros

UV_VERSION_MAJOR
libuv version's major number.

UV_VERSION_MINOR
libuv version's minor number.

UV_VERSION_PATCH
libuv version's patch number.

UV_VERSION_IS_RELEASE
Set to 1 to indicate a release version of libuv, 0 for a development snapshot.

UV_VERSION_SUFFIX
libuv version suffix. Certain development releases such as Release Candidates might have a suffix such as "rc".

UV_VERSION_HEX
Returns the libuv version packed into a single integer. 8 bits are used for each component, with the patch number stored in the 8 least significant bits. E.g. for libuv 1.2.3 this would be 0x010203.

New in version 1.7.0.

6.3.2 Functions

unsigned int **uv_version** (void)
Returns *UV_VERSION_HEX*.

const char* **uv_version_string** (void)
Returns the libuv version number as a string. For non-release versions the version suffix is included.

6.4 `uv_loop_t` — Event loop

The event loop is the central part of libuv's functionality. It takes care of polling for i/o and scheduling callbacks to be run based on different sources of events.

6.4.1 Data types

uv_loop_t
Loop data type.

uv_run_mode
Mode used to run the loop with *uv_run()*.

```
    typedef enum {
        UV_RUN_DEFAULT = 0,
        UV_RUN_ONCE,
        UV_RUN_NOWAIT
    } uv_run_mode;
```

void **(*uv_walk_cb)**(*uv_handle_t* handle*, void* *arg*)
Type definition for callback passed to *uv_walk()*.

Public members

void* **uv_loop_t.data**
Space for user-defined arbitrary data. libuv does not use this field. libuv does, however, initialize it to NULL in *uv_loop_init()*, and it poisons the value (on debug builds) on *uv_loop_close()*.

6.4.2 API

int **uv_loop_init**(*uv_loop_t* loop*)
Initializes the given *uv_loop_t* structure.

int **uv_loop_configure**(*uv_loop_t* loop*, uv_loop_option *option*, ...)
New in version 1.0.2.

Set additional loop options. You should normally call this before the first call to *uv_run()* unless mentioned otherwise.

Returns 0 on success or a UV_E* error code on failure. Be prepared to handle UV_ENOSYS; it means the loop option is not supported by the platform.

Supported options:

- UV_LOOP_BLOCK_SIGNAL: Block a signal when polling for new events. The second argument to *uv_loop_configure()* is the signal number.

  This operation is currently only implemented for SIGPROF signals, to suppress unnecessary wakeups when using a sampling profiler. Requesting other signals will fail with UV_EINVAL.

int **uv_loop_close**(*uv_loop_t* loop*)
Releases all internal loop resources. Call this function only when the loop has finished executing and all open handles and requests have been closed, or it will return UV_EBUSY. After this function returns, the user can free the memory allocated for the loop.

*uv_loop_t** **uv_default_loop**(void)
Returns the initialized default loop. It may return NULL in case of allocation failure.

This function is just a convenient way for having a global loop throughout an application, the default loop is in no way different than the ones initialized with *uv_loop_init()*. As such, the default loop can (and should) be closed with *uv_loop_close()* so the resources associated with it are freed.

int **uv_run**(*uv_loop_t* loop*, *uv_run_mode mode*)
This function runs the event loop. It will act differently depending on the specified mode:

- UV_RUN_DEFAULT: Runs the event loop until there are no more active and referenced handles or requests. Returns non-zero if *uv_stop()* was called and there are still active handles or requests. Returns zero in all other cases.

- UV_RUN_ONCE: Poll for i/o once. Note that this function blocks if there are no pending callbacks. Returns zero when done (no active handles or requests left), or non-zero if more callbacks are expected (meaning you should run the event loop again sometime in the future).
- UV_RUN_NOWAIT: Poll for i/o once but don't block if there are no pending callbacks. Returns zero if done (no active handles or requests left), or non-zero if more callbacks are expected (meaning you should run the event loop again sometime in the future).

int **`uv_loop_alive`**(const *uv_loop_t** *loop*)
Returns non-zero if there are active handles or request in the loop.

void **`uv_stop`**(*uv_loop_t** *loop*)
Stop the event loop, causing `uv_run()` to end as soon as possible. This will happen not sooner than the next loop iteration. If this function was called before blocking for i/o, the loop won't block for i/o on this iteration.

size_t **`uv_loop_size`**(void)
Returns the size of the *uv_loop_t* structure. Useful for FFI binding writers who don't want to know the structure layout.

int **`uv_backend_fd`**(const *uv_loop_t** *loop*)
Get backend file descriptor. Only kqueue, epoll and event ports are supported.

This can be used in conjunction with *uv_run(loop, UV_RUN_NOWAIT)* to poll in one thread and run the event loop's callbacks in another see test/test-embed.c for an example.

Note: Embedding a kqueue fd in another kqueue pollset doesn't work on all platforms. It's not an error to add the fd but it never generates events.

int **`uv_backend_timeout`**(const *uv_loop_t** *loop*)
Get the poll timeout. The return value is in milliseconds, or -1 for no timeout.

uint64_t **`uv_now`**(const *uv_loop_t** *loop*)
Return the current timestamp in milliseconds. The timestamp is cached at the start of the event loop tick, see `uv_update_time()` for details and rationale.

The timestamp increases monotonically from some arbitrary point in time. Don't make assumptions about the starting point, you will only get disappointed.

Note: Use `uv_hrtime()` if you need sub-millisecond granularity.

void **`uv_update_time`**(*uv_loop_t** *loop*)
Update the event loop's concept of "now". Libuv caches the current time at the start of the event loop tick in order to reduce the number of time-related system calls.

You won't normally need to call this function unless you have callbacks that block the event loop for longer periods of time, where "longer" is somewhat subjective but probably on the order of a millisecond or more.

void **`uv_walk`**(*uv_loop_t** *loop*, *uv_walk_cb* *walk_cb*, void* *arg*)
Walk the list of handles: *walk_cb* will be executed with the given *arg*.

6.5 `uv_handle_t` — Base handle

uv_handle_t is the base type for all libuv handle types.

Structures are aligned so that any libuv handle can be cast to *uv_handle_t*. All API functions defined here work with any handle type.

6.5.1 Data types

uv_handle_t
The base libuv handle type.

uv_any_handle
Union of all handle types.

void **(*uv_alloc_cb)**(*uv_handle_t* handle*, size_t *suggested_size*, *uv_buf_t* buf*)
Type definition for callback passed to `uv_read_start()` and `uv_udp_recv_start()`. The user must fill the supplied `uv_buf_t` structure with whatever size, as long as it's > 0. A suggested size (65536 at the moment) is provided, but it doesn't need to be honored. Setting the buffer's length to 0 will trigger a `UV_ENOBUFS` error in the `uv_udp_recv_cb` or `uv_read_cb` callback.

void **(*uv_close_cb)**(*uv_handle_t* handle*)
Type definition for callback passed to `uv_close()`.

Public members

*uv_loop_t** **uv_handle_t.loop**
Pointer to the `uv_loop_t` where the handle is running on. Readonly.

void* **uv_handle_t.data**
Space for user-defined arbitrary data. libuv does not use this field.

6.5.2 API

int **uv_is_active**(const *uv_handle_t* handle*)
Returns non-zero if the handle is active, zero if it's inactive. What "active" means depends on the type of handle:

- A uv_async_t handle is always active and cannot be deactivated, except by closing it with uv_close().
- A uv_pipe_t, uv_tcp_t, uv_udp_t, etc. handle - basically any handle that deals with i/o - is active when it is doing something that involves i/o, like reading, writing, connecting, accepting new connections, etc.
- A uv_check_t, uv_idle_t, uv_timer_t, etc. handle is active when it has been started with a call to uv_check_start(), uv_idle_start(), etc.

Rule of thumb: if a handle of type *uv_foo_t* has a *uv_foo_start()* function, then it's active from the moment that function is called. Likewise, *uv_foo_stop()* deactivates the handle again.

int **uv_is_closing**(const *uv_handle_t* handle*)
Returns non-zero if the handle is closing or closed, zero otherwise.

Note: This function should only be used between the initialization of the handle and the arrival of the close callback.

void **uv_close**(*uv_handle_t* handle*, *uv_close_cb close_cb*)
Request handle to be closed. *close_cb* will be called asynchronously after this call. This MUST be called on each handle before memory is released.

Handles that wrap file descriptors are closed immediately but *close_cb* will still be deferred to the next iteration of the event loop. It gives you a chance to free up any resources associated with the handle.

In-progress requests, like uv_connect_t or uv_write_t, are cancelled and have their callbacks called asynchronously with status=UV_ECANCELED.

void **uv_ref**(*uv_handle_t* handle*)
: Reference the given handle. References are idempotent, that is, if a handle is already referenced calling this function again will have no effect.

  See *Reference counting*.

void **uv_unref**(*uv_handle_t* handle*)
: Un-reference the given handle. References are idempotent, that is, if a handle is not referenced calling this function again will have no effect.

  See *Reference counting*.

int **uv_has_ref**(const *uv_handle_t* handle*)
: Returns non-zero if the handle referenced, zero otherwise.

  See *Reference counting*.

size_t **uv_handle_size**(uv_handle_type *type*)
: Returns the size of the given handle type. Useful for FFI binding writers who don't want to know the structure layout.

6.5.3 Miscellaneous API functions

The following API functions take a `uv_handle_t` argument but they work just for some handle types.

int **uv_send_buffer_size**(*uv_handle_t* handle*, int* *value*)
: Gets or sets the size of the send buffer that the operating system uses for the socket.

  If **value* == 0, it will return the current send buffer size, otherwise it will use **value* to set the new send buffer size.

  This function works for TCP, pipe and UDP handles on Unix and for TCP and UDP handles on Windows.

  Note: Linux will set double the size and return double the size of the original set value.

int **uv_recv_buffer_size**(*uv_handle_t* handle*, int* *value*)
: Gets or sets the size of the receive buffer that the operating system uses for the socket.

  If **value* == 0, it will return the current receive buffer size, otherwise it will use **value* to set the new receive buffer size.

  This function works for TCP, pipe and UDP handles on Unix and for TCP and UDP handles on Windows.

  Note: Linux will set double the size and return double the size of the original set value.

int **uv_fileno**(const *uv_handle_t* handle*, *uv_os_fd_t* fd*)
: Gets the platform dependent file descriptor equivalent.

  The following handles are supported: TCP, pipes, TTY, UDP and poll. Passing any other handle type will fail with *UV_EINVAL*.

  If a handle doesn't have an attached file descriptor yet or the handle itself has been closed, this function will return *UV_EBADF*.

  Warning: Be very careful when using this function. libuv assumes it's in control of the file descriptor so any change to it may lead to malfunction.

6.5.4 Reference counting

The libuv event loop (if run in the default mode) will run until there are no active *and* referenced handles left. The user can force the loop to exit early by unreferencing handles which are active, for example by calling `uv_unref()` after calling `uv_timer_start()`.

A handle can be referenced or unreferenced, the refcounting scheme doesn't use a counter, so both operations are idempotent.

All handles are referenced when active by default, see `uv_is_active()` for a more detailed explanation on what being *active* involves.

6.6 `uv_req_t` — Base request

uv_req_t is the base type for all libuv request types.

Structures are aligned so that any libuv request can be cast to *uv_req_t*. All API functions defined here work with any request type.

6.6.1 Data types

`uv_req_t`
The base libuv request structure.

`uv_any_req`
Union of all request types.

Public members

void* **`uv_req_t.data`**
Space for user-defined arbitrary data. libuv does not use this field.

uv_req_type **`uv_req_t.type`**
Indicated the type of request. Readonly.

```
typedef enum {
    UV_UNKNOWN_REQ = 0,
    UV_REQ,
    UV_CONNECT,
    UV_WRITE,
    UV_SHUTDOWN,
    UV_UDP_SEND,
    UV_FS,
    UV_WORK,
    UV_GETADDRINFO,
    UV_GETNAMEINFO,
    UV_REQ_TYPE_PRIVATE,
    UV_REQ_TYPE_MAX,
} uv_req_type;
```

6.6.2 API

int **`uv_cancel`**(*uv_req_t* req*)
Cancel a pending request. Fails if the request is executing or has finished executing.

Returns 0 on success, or an error code < 0 on failure.

Only cancellation of `uv_fs_t`, `uv_getaddrinfo_t`, `uv_getnameinfo_t` and `uv_work_t` requests is currently supported.

Cancelled requests have their callbacks invoked some time in the future. It's **not** safe to free the memory associated with the request until the callback is called.

Here is how cancellation is reported to the callback:

- A `uv_fs_t` request has its req->result field set to *UV_ECANCELED*.
- A `uv_work_t`, `uv_getaddrinfo_t` or c:type:*uv_getnameinfo_t* request has its callback invoked with status == *UV_ECANCELED*.

size_t **`uv_req_size`** (uv_req_type *type*)
Returns the size of the given request type. Useful for FFI binding writers who don't want to know the structure layout.

6.7 `uv_timer_t` — Timer handle

Timer handles are used to schedule callbacks to be called in the future.

6.7.1 Data types

`uv_timer_t`
Timer handle type.

void **`(*uv_timer_cb)`** (*uv_timer_t* handle*)
Type definition for callback passed to `uv_timer_start()`.

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.7.2 API

int **`uv_timer_init`** (*uv_loop_t* loop*, *uv_timer_t* handle*)
Initialize the handle.

int **`uv_timer_start`** (*uv_timer_t* handle*, *uv_timer_cb cb*, uint64_t *timeout*, uint64_t *repeat*)
Start the timer. *timeout* and *repeat* are in milliseconds.

If *timeout* is zero, the callback fires on the next event loop iteration. If *repeat* is non-zero, the callback fires first after *timeout* milliseconds and then repeatedly after *repeat* milliseconds.

int **`uv_timer_stop`** (*uv_timer_t* handle*)
Stop the timer, the callback will not be called anymore.

int **`uv_timer_again`** (*uv_timer_t* handle*)
Stop the timer, and if it is repeating restart it using the repeat value as the timeout. If the timer has never been started before it returns UV_EINVAL.

void **uv_timer_set_repeat** (*uv_timer_t* handle*, uint64_t *repeat*)

Set the repeat interval value in milliseconds. The timer will be scheduled to run on the given interval, regardless of the callback execution duration, and will follow normal timer semantics in the case of a time-slice overrun.

For example, if a 50ms repeating timer first runs for 17ms, it will be scheduled to run again 33ms later. If other tasks consume more than the 33ms following the first timer callback, then the callback will run as soon as possible.

Note: If the repeat value is set from a timer callback it does not immediately take effect. If the timer was non-repeating before, it will have been stopped. If it was repeating, then the old repeat value will have been used to schedule the next timeout.

uint64_t **uv_timer_get_repeat** (const *uv_timer_t* handle*)

Get the timer repeat value.

See also:

The `uv_handle_t` API functions also apply.

6.8 `uv_prepare_t` — Prepare handle

Prepare handles will run the given callback once per loop iteration, right before polling for i/o.

6.8.1 Data types

uv_prepare_t

Prepare handle type.

void **(*uv_prepare_cb)** (*uv_prepare_t* handle*)

Type definition for callback passed to `uv_prepare_start()`.

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.8.2 API

int **uv_prepare_init** (*uv_loop_t* loop*, *uv_prepare_t* prepare*)

Initialize the handle.

int **uv_prepare_start** (*uv_prepare_t* prepare*, *uv_prepare_cb cb*)

Start the handle with the given callback.

int **uv_prepare_stop** (*uv_prepare_t* prepare*)

Stop the handle, the callback will no longer be called.

See also:

The `uv_handle_t` API functions also apply.

6.9 `uv_check_t` — Check handle

Check handles will run the given callback once per loop iteration, right after polling for i/o.

6.9.1 Data types

`uv_check_t`
Check handle type.

void **`(*uv_check_cb)`** (*uv_check_t* handle*)
Type definition for callback passed to `uv_check_start()`.

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.9.2 API

int **`uv_check_init`** (*uv_loop_t* loop*, *uv_check_t* check*)
Initialize the handle.

int **`uv_check_start`** (*uv_check_t* check*, *uv_check_cb cb*)
Start the handle with the given callback.

int **`uv_check_stop`** (*uv_check_t* check*)
Stop the handle, the callback will no longer be called.

See also:

The `uv_handle_t` API functions also apply.

6.10 `uv_idle_t` — Idle handle

Idle handles will run the given callback once per loop iteration, right before the `uv_prepare_t` handles.

Note: The notable difference with prepare handles is that when there are active idle handles, the loop will perform a zero timeout poll instead of blocking for i/o.

Warning: Despite the name, idle handles will get their callbacks called on every loop iteration, not when the loop is actually "idle".

6.10.1 Data types

`uv_idle_t`
Idle handle type.

void **(*uv_idle_cb)** (*uv_idle_t* handle*)
Type definition for callback passed to `uv_idle_start()`.

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.10.2 API

int **uv_idle_init** (*uv_loop_t* loop*, *uv_idle_t* idle*)
Initialize the handle.

int **uv_idle_start** (*uv_idle_t* idle*, *uv_idle_cb cb*)
Start the handle with the given callback.

int **uv_idle_stop** (*uv_idle_t* idle*)
Stop the handle, the callback will no longer be called.

See also:

The `uv_handle_t` API functions also apply.

6.11 `uv_async_t` — Async handle

Async handles allow the user to "wakeup" the event loop and get a callback called from another thread.

6.11.1 Data types

uv_async_t
Async handle type.

void **(*uv_async_cb)** (*uv_async_t* handle*)
Type definition for callback passed to `uv_async_init()`.

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.11.2 API

int **uv_async_init** (*uv_loop_t* loop*, *uv_async_t* async*, *uv_async_cb async_cb*)
Initialize the handle. A NULL callback is allowed.

Note: Unlike other handle initialization functions, it immediately starts the handle.

int **uv_async_send**(*uv_async_t* async*)

Wakeup the event loop and call the async handle's callback.

Note: It's safe to call this function from any thread. The callback will be called on the loop thread.

Warning: libuv will coalesce calls to `uv_async_send()`, that is, not every call to it will yield an execution of the callback. For example: if `uv_async_send()` is called 5 times in a row before the callback is called, the callback will only be called once. If `uv_async_send()` is called again after the callback was called, it will be called again.

See also:

The `uv_handle_t` API functions also apply.

6.12 `uv_poll_t` — Poll handle

Poll handles are used to watch file descriptors for readability, writability and disconnection similar to the purpose of poll(2).

The purpose of poll handles is to enable integrating external libraries that rely on the event loop to signal it about the socket status changes, like c-ares or libssh2. Using uv_poll_t for any other purpose is not recommended; `uv_tcp_t`, `uv_udp_t`, etc. provide an implementation that is faster and more scalable than what can be achieved with `uv_poll_t`, especially on Windows.

It is possible that poll handles occasionally signal that a file descriptor is readable or writable even when it isn't. The user should therefore always be prepared to handle EAGAIN or equivalent when it attempts to read from or write to the fd.

It is not okay to have multiple active poll handles for the same socket, this can cause libuv to busyloop or otherwise malfunction.

The user should not close a file descriptor while it is being polled by an active poll handle. This can cause the handle to report an error, but it might also start polling another socket. However the fd can be safely closed immediately after a call to `uv_poll_stop()` or `uv_close()`.

Note: On windows only sockets can be polled with poll handles. On Unix any file descriptor that would be accepted by poll(2) can be used.

6.12.1 Data types

uv_poll_t

Poll handle type.

void **(*uv_poll_cb)**(*uv_poll_t* handle*, int *status*, int *events*)

Type definition for callback passed to `uv_poll_start()`.

uv_poll_event

Poll event types

```
enum uv_poll_event {
    UV_READABLE = 1,
    UV_WRITABLE = 2,
    UV_DISCONNECT = 4
};
```

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.12.2 API

int **`uv_poll_init`**(*`uv_loop_t`* loop*, *`uv_poll_t`* handle*, int *fd*)
Initialize the handle using a file descriptor.

Changed in version 1.2.2: the file descriptor is set to non-blocking mode.

int **`uv_poll_init_socket`**(*`uv_loop_t`* loop*, *`uv_poll_t`* handle*, *`uv_os_sock_t` socket*)
Initialize the handle using a socket descriptor. On Unix this is identical to `uv_poll_init()`. On windows it takes a SOCKET handle.

Changed in version 1.2.2: the socket is set to non-blocking mode.

int **`uv_poll_start`**(*`uv_poll_t`* handle*, int *events*, *`uv_poll_cb` cb*)
Starts polling the file descriptor. *events* is a bitmask consisting made up of UV_READABLE, UV_WRITABLE and UV_DISCONNECT. As soon as an event is detected the callback will be called with *status* set to 0, and the detected events set on the *events* field.

The UV_DISCONNECT event is optional in the sense that it may not be reported and the user is free to ignore it, but it can help optimize the shutdown path because an extra read or write call might be avoided.

If an error happens while polling, *status* will be < 0 and corresponds with one of the UV_E* error codes (see *Error handling*). The user should not close the socket while the handle is active. If the user does that anyway, the callback *may* be called reporting an error status, but this is **not** guaranteed.

Note: Calling `uv_poll_start()` on a handle that is already active is fine. Doing so will update the events mask that is being watched for.

Changed in version 1.9.0: Added the UV_DISCONNECT event.

int **`uv_poll_stop`**(*`uv_poll_t`* poll*)
Stop polling the file descriptor, the callback will no longer be called.

See also:

The `uv_handle_t` API functions also apply.

6.13 `uv_signal_t` — Signal handle

Signal handles implement Unix style signal handling on a per-event loop bases.

Reception of some signals is emulated on Windows:

- SIGINT is normally delivered when the user presses CTRL+C. However, like on Unix, it is not generated when terminal raw mode is enabled.

- SIGBREAK is delivered when the user pressed CTRL + BREAK.
- SIGHUP is generated when the user closes the console window. On SIGHUP the program is given approximately 10 seconds to perform cleanup. After that Windows will unconditionally terminate it.
- SIGWINCH is raised whenever libuv detects that the console has been resized. SIGWINCH is emulated by libuv when the program uses a `uv_tty_t` handle to write to the console. SIGWINCH may not always be delivered in a timely manner; libuv will only detect size changes when the cursor is being moved. When a readable `uv_tty_t` handle is used in raw mode, resizing the console buffer will also trigger a SIGWINCH signal.

Watchers for other signals can be successfully created, but these signals are never received. These signals are: *SIGILL*, *SIGABRT*, *SIGFPE*, *SIGSEGV*, *SIGTERM* and *SIGKILL.*

Calls to raise() or abort() to programmatically raise a signal are not detected by libuv; these will not trigger a signal watcher.

Note: On Linux SIGRT0 and SIGRT1 (signals 32 and 33) are used by the NPTL pthreads library to manage threads. Installing watchers for those signals will lead to unpredictable behavior and is strongly discouraged. Future versions of libuv may simply reject them.

6.13.1 Data types

`uv_signal_t`

Signal handle type.

void **`(*uv_signal_cb)`** (*uv_signal_t* handle*, int *signum*)

Type definition for callback passed to `uv_signal_start()`.

Public members

int **`uv_signal_t.signum`**

Signal being monitored by this handle. Readonly.

See also:

The `uv_handle_t` members also apply.

6.13.2 API

int **`uv_signal_init`** (*uv_loop_t* loop*, *uv_signal_t* signal*)

Initialize the handle.

int **`uv_signal_start`** (*uv_signal_t* signal*, *uv_signal_cb cb*, int *signum*)

Start the handle with the given callback, watching for the given signal.

int **`uv_signal_stop`** (*uv_signal_t* signal*)

Stop the handle, the callback will no longer be called.

See also:

The `uv_handle_t` API functions also apply.

6.14 `uv_process_t` — Process handle

Process handles will spawn a new process and allow the user to control it and establish communication channels with it using streams.

6.14.1 Data types

`uv_process_t`
Process handle type.

`uv_process_options_t`
Options for spawning the process (passed to `uv_spawn()`.

```
typedef struct uv_process_options_s {
    uv_exit_cb exit_cb;
    const char* file;
    char** args;
    char** env;
    const char* cwd;
    unsigned int flags;
    int stdio_count;
    uv_stdio_container_t* stdio;
    uv_uid_t uid;
    uv_gid_t gid;
} uv_process_options_t;
```

void **`(*uv_exit_cb)`** (*uv_process_t**, int64_t *exit_status*, int *term_signal*)
Type definition for callback passed in `uv_process_options_t` which will indicate the exit status and the signal that caused the process to terminate, if any.

`uv_process_flags`
Flags to be set on the flags field of `uv_process_options_t`.

```
enum uv_process_flags {
    /*
    * Set the child process' user id.
    */
    UV_PROCESS_SETUID = (1 << 0),
    /*
    * Set the child process' group id.
    */
    UV_PROCESS_SETGID = (1 << 1),
    /*
    * Do not wrap any arguments in quotes, or perform any other escaping, when
    * converting the argument list into a command line string. This option is
    * only meaningful on Windows systems. On Unix it is silently ignored.
    */
    UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS = (1 << 2),
    /*
    * Spawn the child process in a detached state - this will make it a process
    * group leader, and will effectively enable the child to keep running after
    * the parent exits. Note that the child process will still keep the
    * parent's event loop alive unless the parent process calls uv_unref() on
    * the child's process handle.
    */
    UV_PROCESS_DETACHED = (1 << 3),
    /*
```

```
        * Hide the subprocess console window that would normally be created. This
        * option is only meaningful on Windows systems. On Unix it is silently
        * ignored.
        */
        UV_PROCESS_WINDOWS_HIDE = (1 << 4)
    };
```

uv_stdio_container_t

Container for each stdio handle or fd passed to a child process.

```
    typedef struct uv_stdio_container_s {
        uv_stdio_flags flags;
        union {
            uv_stream_t* stream;
            int fd;
        } data;
    } uv_stdio_container_t;
```

uv_stdio_flags

Flags specifying how a stdio should be transmitted to the child process.

```
    typedef enum {
        UV_IGNORE = 0x00,
        UV_CREATE_PIPE = 0x01,
        UV_INHERIT_FD = 0x02,
        UV_INHERIT_STREAM = 0x04,
        /*
        * When UV_CREATE_PIPE is specified, UV_READABLE_PIPE and UV_WRITABLE_PIPE
        * determine the direction of flow, from the child process' perspective. Both
        * flags may be specified to create a duplex data stream.
        */
        UV_READABLE_PIPE = 0x10,
        UV_WRITABLE_PIPE = 0x20
    } uv_stdio_flags;
```

Public members

uv_process_t.pid

The PID of the spawned process. It's set after calling `uv_spawn()`.

Note: The `uv_handle_t` members also apply.

uv_process_options_t.exit_cb

Callback called after the process exits.

uv_process_options_t.file

Path pointing to the program to be executed.

uv_process_options_t.args

Command line arguments. args[0] should be the path to the program. On Windows this uses *CreateProcess* which concatenates the arguments into a string this can cause some strange errors. See the `UV_PROCESS_WINDOWS_VERBATIM_ARGUMENTS` flag on `uv_process_flags`.

uv_process_options_t.env

Environment for the new process. If NULL the parents environment is used.

`uv_process_options_t.cwd`
Current working directory for the subprocess.

`uv_process_options_t.flags`
Various flags that control how `uv_spawn()` behaves. See `uv_process_flags`.

`uv_process_options_t.stdio_count`

`uv_process_options_t.stdio`
The *stdio* field points to an array of `uv_stdio_container_t` structs that describe the file descriptors that will be made available to the child process. The convention is that stdio[0] points to stdin, fd 1 is used for stdout, and fd 2 is stderr.

Note: On Windows file descriptors greater than 2 are available to the child process only if the child processes uses the MSVCRT runtime.

`uv_process_options_t.uid`

`uv_process_options_t.gid`
Libuv can change the child process' user/group id. This happens only when the appropriate bits are set in the flags fields.

Note: This is not supported on Windows, `uv_spawn()` will fail and set the error to `UV_ENOTSUP`.

`uv_stdio_container_t.flags`
Flags specifying how the stdio container should be passed to the child. See `uv_stdio_flags`.

`uv_stdio_container_t.data`
Union containing either the stream or fd to be passed on to the child process.

6.14.2 API

void **`uv_disable_stdio_inheritance`**(void)
Disables inheritance for file descriptors / handles that this process inherited from its parent. The effect is that child processes spawned by this process don't accidentally inherit these handles.

It is recommended to call this function as early in your program as possible, before the inherited file descriptors can be closed or duplicated.

Note: This function works on a best-effort basis: there is no guarantee that libuv can discover all file descriptors that were inherited. In general it does a better job on Windows than it does on Unix.

int **`uv_spawn`**(*uv_loop_t* loop*, *uv_process_t* handle*, const *uv_process_options_t* options*)
Initializes the process handle and starts the process. If the process is successfully spawned, this function will return 0. Otherwise, the negative error code corresponding to the reason it couldn't spawn is returned.

Possible reasons for failing to spawn would include (but not be limited to) the file to execute not existing, not having permissions to use the setuid or setgid specified, or not having enough memory to allocate for the new process.

int **`uv_process_kill`**(*uv_process_t* handle*, int *signum*)
Sends the specified signal to the given process handle. Check the documentation on *uv_signal_t — Signal handle* for signal support, specially on Windows.

int **uv_kill** (int *pid*, int *signum*)
Sends the specified signal to the given PID. Check the documentation on *uv_signal_t — Signal handle* for signal support, specially on Windows.

See also:

The `uv_handle_t` API functions also apply.

6.15 `uv_stream_t` — Stream handle

Stream handles provide an abstraction of a duplex communication channel. `uv_stream_t` is an abstract type, libuv provides 3 stream implementations in the for of `uv_tcp_t`, `uv_pipe_t` and `uv_tty_t`.

6.15.1 Data types

`uv_stream_t`
Stream handle type.

`uv_connect_t`
Connect request type.

`uv_shutdown_t`
Shutdown request type.

`uv_write_t`
Write request type.

void **(*uv_read_cb)** (*uv_stream_t* stream*, ssize_t *nread*, const *uv_buf_t* buf*)
Callback called when data was read on a stream.

nread is > 0 if there is data available or < 0 on error. When we've reached EOF, *nread* will be set to `UV_EOF`. When *nread* < 0, the *buf* parameter might not point to a valid buffer; in that case *buf.len* and *buf.base* are both set to 0.

Note: *nread* might be 0, which does *not* indicate an error or EOF. This is equivalent to `EAGAIN` or `EWOULDBLOCK` under `read(2)`.

The callee is responsible for stopping closing the stream when an error happens by calling `uv_read_stop()` or `uv_close()`. Trying to read from the stream again is undefined.

The callee is responsible for freeing the buffer, libuv does not reuse it. The buffer may be a null buffer (where buf->base=NULL and buf->len=0) on error.

void **(*uv_write_cb)** (*uv_write_t* req*, int *status*)
Callback called after data was written on a stream. *status* will be 0 in case of success, < 0 otherwise.

void **(*uv_connect_cb)** (*uv_connect_t* req*, int *status*)
Callback called after a connection started by `uv_connect()` is done. *status* will be 0 in case of success, < 0 otherwise.

void **(*uv_shutdown_cb)** (*uv_shutdown_t* req*, int *status*)
Callback called after s shutdown request has been completed. *status* will be 0 in case of success, < 0 otherwise.

void **(*uv_connection_cb)** (*uv_stream_t* server*, int *status*)
Callback called when a stream server has received an incoming connection. The user can accept the connection by calling `uv_accept()`. *status* will be 0 in case of success, < 0 otherwise.

Public members

size_t **uv_stream_t.write_queue_size**
: Contains the amount of queued bytes waiting to be sent. Readonly.

*uv_stream_t** **uv_connect_t.handle**
: Pointer to the stream where this connection request is running.

*uv_stream_t** **uv_shutdown_t.handle**
: Pointer to the stream where this shutdown request is running.

*uv_stream_t** **uv_write_t.handle**
: Pointer to the stream where this write request is running.

*uv_stream_t** **uv_write_t.send_handle**
: Pointer to the stream being sent using this write request..

See also:

The `uv_handle_t` members also apply.

6.15.2 API

int **uv_shutdown** (*uv_shutdown_t* req*, *uv_stream_t* handle*, *uv_shutdown_cb cb*)
: Shutdown the outgoing (write) side of a duplex stream. It waits for pending write requests to complete. The *handle* should refer to a initialized stream. *req* should be an uninitialized shutdown request struct. The *cb* is called after shutdown is complete.

int **uv_listen** (*uv_stream_t* stream*, int *backlog*, *uv_connection_cb cb*)
: Start listening for incoming connections. *backlog* indicates the number of connections the kernel might queue, same as listen(2). When a new incoming connection is received the `uv_connection_cb` callback is called.

int **uv_accept** (*uv_stream_t* server*, *uv_stream_t* client*)
: This call is used in conjunction with `uv_listen()` to accept incoming connections. Call this function after receiving a `uv_connection_cb` to accept the connection. Before calling this function the client handle must be initialized. < 0 return value indicates an error.

  When the `uv_connection_cb` callback is called it is guaranteed that this function will complete successfully the first time. If you attempt to use it more than once, it may fail. It is suggested to only call this function once per `uv_connection_cb` call.

  Note: *server* and *client* must be handles running on the same loop.

int **uv_read_start** (*uv_stream_t* stream*, *uv_alloc_cb alloc_cb*, *uv_read_cb read_cb*)
: Read data from an incoming stream. The `uv_read_cb` callback will be made several times until there is no more data to read or `uv_read_stop()` is called.

int **uv_read_stop** (*uv_stream_t**)
: Stop reading data from the stream. The `uv_read_cb` callback will no longer be called.

  This function is idempotent and may be safely called on a stopped stream.

int **uv_write** (*uv_write_t* req*, *uv_stream_t* handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, *uv_write_cb cb*)
: Write data to stream. Buffers are written in order. Example:

```
void cb(uv_write_t* req, int status) {
    /* Logic which handles the write result */
}
```

```
uv_buf_t a[] = {
    { .base = "1", .len = 1 },
    { .base = "2", .len = 1 }
};

uv_buf_t b[] = {
    { .base = "3", .len = 1 },
    { .base = "4", .len = 1 }
};

uv_write_t req1;
uv_write_t req2;

/* writes "1234" */
uv_write(&req1, stream, a, 2, cb);
uv_write(&req2, stream, b, 2, cb);
```

int **uv_write2**(*uv_write_t* req*, *uv_stream_t* handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, *uv_stream_t* send_handle*, *uv_write_cb cb*)

Extended write function for sending handles over a pipe. The pipe must be initialized with *ipc* == 1.

Note: *send_handle* must be a TCP socket or pipe, which is a server or a connection (listening or connected state). Bound sockets or pipes will be assumed to be servers.

int **uv_try_write**(*uv_stream_t* handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*)

Same as `uv_write()`, but won't queue a write request if it can't be completed immediately.

Will return either:

- > 0: number of bytes written (can be less than the supplied buffer size).
- < 0: negative error code (`UV_EAGAIN` is returned if no data can be sent immediately).

int **uv_is_readable**(const *uv_stream_t* handle*)

Returns 1 if the stream is readable, 0 otherwise.

int **uv_is_writable**(const *uv_stream_t* handle*)

Returns 1 if the stream is writable, 0 otherwise.

int **uv_stream_set_blocking**(*uv_stream_t* handle*, int *blocking*)

Enable or disable blocking mode for a stream.

When blocking mode is enabled all writes complete synchronously. The interface remains unchanged otherwise, e.g. completion or failure of the operation will still be reported through a callback which is made asynchronously.

Warning: Relying too much on this API is not recommended. It is likely to change significantly in the future.
Currently only works on Windows for `uv_pipe_t` handles. On UNIX platforms, all `uv_stream_t` handles are supported.
Also libuv currently makes no ordering guarantee when the blocking mode is changed after write requests have already been submitted. Therefore it is recommended to set the blocking mode immediately after opening or creating the stream.

Changed in version 1.4.0: UNIX implementation added.

See also:

The `uv_handle_t` API functions also apply.

6.16 `uv_tcp_t` — TCP handle

TCP handles are used to represent both TCP streams and servers.

`uv_tcp_t` is a 'subclass' of `uv_stream_t`.

6.16.1 Data types

`uv_tcp_t`
TCP handle type.

Public members

N/A

See also:

The `uv_stream_t` members also apply.

6.16.2 API

int **`uv_tcp_init`**(*uv_loop_t* loop*, *uv_tcp_t* handle*)
Initialize the handle. No socket is created as of yet.

int **`uv_tcp_init_ex`**(*uv_loop_t* loop*, *uv_tcp_t* handle*, unsigned int *flags*)
Initialize the handle with the specified flags. At the moment only the lower 8 bits of the *flags* parameter are used as the socket domain. A socket will be created for the given domain. If the specified domain is `AF_UNSPEC` no socket is created, just like `uv_tcp_init()`.

New in version 1.7.0.

int **`uv_tcp_open`**(*uv_tcp_t* handle*, *uv_os_sock_t sock*)
Open an existing file descriptor or SOCKET as a TCP handle.

Changed in version 1.2.1: the file descriptor is set to non-blocking mode.

Note: The passed file descriptor or SOCKET is not checked for its type, but it's required that it represents a valid stream socket.

int **`uv_tcp_nodelay`**(*uv_tcp_t* handle*, int *enable*)
Enable / disable Nagle's algorithm.

int **`uv_tcp_keepalive`**(*uv_tcp_t* handle*, int *enable*, unsigned int *delay*)
Enable / disable TCP keep-alive. *delay* is the initial delay in seconds, ignored when *enable* is zero.

int **`uv_tcp_simultaneous_accepts`**(*uv_tcp_t* handle*, int *enable*)
Enable / disable simultaneous asynchronous accept requests that are queued by the operating system when listening for new TCP connections.

This setting is used to tune a TCP server for the desired performance. Having simultaneous accepts can significantly improve the rate of accepting connections (which is why it is enabled by default) but may lead to uneven load distribution in multi-process setups.

int **uv_tcp_bind**(*uv_tcp_t* handle*, const struct sockaddr* *addr*, unsigned int *flags*)

Bind the handle to an address and port. *addr* should point to an initialized `struct sockaddr_in` or `struct sockaddr_in6`.

When the port is already taken, you can expect to see an `UV_EADDRINUSE` error from either `uv_tcp_bind()`, `uv_listen()` or `uv_tcp_connect()`. That is, a successful call to this function does not guarantee that the call to `uv_listen()` or `uv_tcp_connect()` will succeed as well.

flags can contain `UV_TCP_IPV6ONLY`, in which case dual-stack support is disabled and only IPv6 is used.

int **uv_tcp_getsockname**(const *uv_tcp_t* handle*, struct sockaddr* *name*, int* *namelen*)

Get the current address to which the handle is bound. *addr* must point to a valid and big enough chunk of memory, `struct sockaddr_storage` is recommended for IPv4 and IPv6 support.

int **uv_tcp_getpeername**(const *uv_tcp_t* handle*, struct sockaddr* *name*, int* *namelen*)

Get the address of the peer connected to the handle. *addr* must point to a valid and big enough chunk of memory, `struct sockaddr_storage` is recommended for IPv4 and IPv6 support.

int **uv_tcp_connect**(*uv_connect_t* req*, *uv_tcp_t* handle*, const struct sockaddr* *addr*, *uv_connect_cb cb*)

Establish an IPv4 or IPv6 TCP connection. Provide an initialized TCP handle and an uninitialized `uv_connect_t`. *addr* should point to an initialized `struct sockaddr_in` or `struct sockaddr_in6`.

The callback is made when the connection has been established or when a connection error happened.

See also:

The `uv_stream_t` API functions also apply.

6.17 `uv_pipe_t` — Pipe handle

Pipe handles provide an abstraction over local domain sockets on Unix and named pipes on Windows.

`uv_pipe_t` is a 'subclass' of `uv_stream_t`.

6.17.1 Data types

`uv_pipe_t`

Pipe handle type.

Public members

N/A

See also:

The `uv_stream_t` members also apply.

6.17.2 API

int **uv_pipe_init**(*uv_loop_t* loop*, *uv_pipe_t* handle*, int *ipc*)

Initialize a pipe handle. The *ipc* argument is a boolean to indicate if this pipe will be used for handle passing between processes.

int **uv_pipe_open**(*uv_pipe_t* handle*, *uv_file file*)

Open an existing file descriptor or HANDLE as a pipe.

Changed in version 1.2.1: the file descriptor is set to non-blocking mode.

Note: The passed file descriptor or HANDLE is not checked for its type, but it's required that it represents a valid pipe.

int **uv_pipe_bind**(*uv_pipe_t* handle*, const char* *name*)

Bind the pipe to a file path (Unix) or a name (Windows).

Note: Paths on Unix get truncated to `sizeof(sockaddr_un.sun_path)` bytes, typically between 92 and 108 bytes.

void **uv_pipe_connect**(*uv_connect_t* req*, *uv_pipe_t* handle*, const char* *name*, *uv_connect_cb cb*)

Connect to the Unix domain socket or the named pipe.

Note: Paths on Unix get truncated to `sizeof(sockaddr_un.sun_path)` bytes, typically between 92 and 108 bytes.

int **uv_pipe_getsockname**(const *uv_pipe_t* handle*, char* *buffer*, size_t* *size*)

Get the name of the Unix domain socket or the named pipe.

A preallocated buffer must be provided. The size parameter holds the length of the buffer and it's set to the number of bytes written to the buffer on output. If the buffer is not big enough `UV_ENOBUFS` will be returned and len will contain the required size.

Changed in version 1.3.0: the returned length no longer includes the terminating null byte, and the buffer is not null terminated.

int **uv_pipe_getpeername**(const *uv_pipe_t* handle*, char* *buffer*, size_t* *size*)

Get the name of the Unix domain socket or the named pipe to which the handle is connected.

A preallocated buffer must be provided. The size parameter holds the length of the buffer and it's set to the number of bytes written to the buffer on output. If the buffer is not big enough `UV_ENOBUFS` will be returned and len will contain the required size.

New in version 1.3.0.

void **uv_pipe_pending_instances**(*uv_pipe_t* handle*, int *count*)

Set the number of pending pipe instance handles when the pipe server is waiting for connections.

Note: This setting applies to Windows only.

int **uv_pipe_pending_count**(*uv_pipe_t* handle*)

uv_handle_type **uv_pipe_pending_type**(*uv_pipe_t* handle*)

Used to receive handles over IPC pipes.

First - call `uv_pipe_pending_count()`, if it's > 0 then initialize a handle of the given *type*, returned by `uv_pipe_pending_type()` and call `uv_accept(pipe, handle)`.

See also:

The `uv_stream_t` API functions also apply.

6.18 `uv_tty_t` — TTY handle

TTY handles represent a stream for the console.

`uv_tty_t` is a 'subclass' of *`uv_stream_t`*.

6.18.1 Data types

`uv_tty_t`
TTY handle type.

`uv_tty_mode_t`
New in version 1.2.0.

TTY mode type:

```
typedef enum {
    /* Initial/normal terminal mode */
    UV_TTY_MODE_NORMAL,
    /* Raw input mode (On Windows, ENABLE_WINDOW_INPUT is also enabled) */
    UV_TTY_MODE_RAW,
    /* Binary-safe I/O mode for IPC (Unix-only) */
    UV_TTY_MODE_IO
} uv_tty_mode_t;
```

Public members

N/A

See also:

The *`uv_stream_t`* members also apply.

6.18.2 API

int **`uv_tty_init`**(*uv_loop_t* loop*, *uv_tty_t* handle*, *uv_file fd*, int *readable*)
Initialize a new TTY stream with the given file descriptor. Usually the file descriptor will be:

- 0 = stdin
- 1 = stdout
- 2 = stderr

readable, specifies if you plan on calling *`uv_read_start()`* with this stream. stdin is readable, stdout is not.

On Unix this function will determine the path of the fd of the terminal using ttyname_r(3), open it, and use it if the passed file descriptor refers to a TTY. This lets libuv put the tty in non-blocking mode without affecting other processes that share the tty.

This function is not thread safe on systems that don't support ioctl TIOCGPTN or TIOCPTYGNAME, for instance OpenBSD and Solaris.

Note: If reopening the TTY fails, libuv falls back to blocking writes for non-readable TTY streams.

Changed in version 1.9.0:: the path of the TTY is determined by ttyname_r(3). In earlier versions libuv opened */dev/tty* instead.

Changed in version 1.5.0:: trying to initialize a TTY stream with a file descriptor that refers to a file returns *UV_EINVAL* on UNIX.

int **uv_tty_set_mode**(*uv_tty_t* handle*, *uv_tty_mode_t mode*)

Changed in version 1.2.0:: the mode is specified as a `uv_tty_mode_t` value.

Set the TTY using the specified terminal mode.

int **uv_tty_reset_mode**(void)

To be called when the program exits. Resets TTY settings to default values for the next process to take over.

This function is async signal-safe on Unix platforms but can fail with error code `UV_EBUSY` if you call it when execution is inside `uv_tty_set_mode()`.

int **uv_tty_get_winsize**(*uv_tty_t* handle*, int* *width*, int* *height*)

Gets the current Window size. On success it returns 0.

See also:

The `uv_stream_t` API functions also apply.

6.19 `uv_udp_t` — UDP handle

UDP handles encapsulate UDP communication for both clients and servers.

6.19.1 Data types

`uv_udp_t`

UDP handle type.

`uv_udp_send_t`

UDP send request type.

`uv_udp_flags`

Flags used in `uv_udp_bind()` and `uv_udp_recv_cb`..

```
enum uv_udp_flags {
    /* Disables dual stack mode. */
    UV_UDP_IPV6ONLY = 1,
    /*
    * Indicates message was truncated because read buffer was too small. The
    * remainder was discarded by the OS. Used in uv_udp_recv_cb.
    */
    UV_UDP_PARTIAL = 2,
    /*
    * Indicates if SO_REUSEADDR will be set when binding the handle in
    * uv_udp_bind.
    * This sets the SO_REUSEPORT socket flag on the BSDs and OS X. On other
    * Unix platforms, it sets the SO_REUSEADDR flag. What that means is that
    * multiple threads or processes can bind to the same address without error
    * (provided they all set the flag) but only the last one to bind will receive
    * any traffic, in effect "stealing" the port from the previous listener.
    */
    UV_UDP_REUSEADDR = 4
};
```

void **(*uv_udp_send_cb)** (*uv_udp_send_t* req*, int *status*)
Type definition for callback passed to `uv_udp_send()`, which is called after the data was sent.

void **(*uv_udp_recv_cb)** (*uv_udp_t* handle*, ssize_t *nread*, const *uv_buf_t* buf*, const struct sockaddr* *addr*, unsigned *flags*)
Type definition for callback passed to `uv_udp_recv_start()`, which is called when the endpoint receives data.

- *handle*: UDP handle
- *nread*: Number of bytes that have been received. 0 if there is no more data to read. You may discard or repurpose the read buffer. Note that 0 may also mean that an empty datagram was received (in this case *addr* is not NULL). < 0 if a transmission error was detected.
- *buf*: `uv_buf_t` with the received data.
- *addr*: `struct sockaddr*` containing the address of the sender. Can be NULL. Valid for the duration of the callback only.
- *flags*: One or more or'ed UV_UDP_* constants. Right now only `UV_UDP_PARTIAL` is used.

Note: The receive callback will be called with *nread* == 0 and *addr* == NULL when there is nothing to read, and with *nread* == 0 and *addr* != NULL when an empty UDP packet is received.

uv_membership
Membership type for a multicast address.

```
typedef enum {
    UV_LEAVE_GROUP = 0,
    UV_JOIN_GROUP
} uv_membership;
```

Public members

size_t **uv_udp_t.send_queue_size**
Number of bytes queued for sending. This field strictly shows how much information is currently queued.

size_t **uv_udp_t.send_queue_count**
Number of send requests currently in the queue awaiting to be processed.

*uv_udp_t** **uv_udp_send_t.handle**
UDP handle where this send request is taking place.

See also:

The `uv_handle_t` members also apply.

6.19.2 API

int **uv_udp_init** (*uv_loop_t* loop*, *uv_udp_t* handle*)
Initialize a new UDP handle. The actual socket is created lazily. Returns 0 on success.

int **uv_udp_init_ex** (*uv_loop_t* loop*, *uv_udp_t* handle*, unsigned int *flags*)
Initialize the handle with the specified flags. At the moment the lower 8 bits of the *flags* parameter are used as the socket domain. A socket will be created for the given domain. If the specified domain is `AF_UNSPEC` no socket is created, just like `uv_udp_init()`.

New in version 1.7.0.

int **uv_udp_open** (*uv_udp_t* handle*, *uv_os_sock_t sock*)

Opens an existing file descriptor or Windows SOCKET as a UDP handle.

Unix only: The only requirement of the *sock* argument is that it follows the datagram contract (works in unconnected mode, supports sendmsg()/recvmsg(), etc). In other words, other datagram-type sockets like raw sockets or netlink sockets can also be passed to this function.

Changed in version 1.2.1: the file descriptor is set to non-blocking mode.

Note: The passed file descriptor or SOCKET is not checked for its type, but it's required that it represents a valid datagram socket.

int **uv_udp_bind** (*uv_udp_t* handle*, const struct sockaddr* *addr*, unsigned int *flags*)

Bind the UDP handle to an IP address and port.

Parameters

- **handle** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **addr** – *struct sockaddr_in* or *struct sockaddr_in6* with the address and port to bind to.
- **flags** – Indicate how the socket will be bound, `UV_UDP_IPV6ONLY` and `UV_UDP_REUSEADDR` are supported.

Returns 0 on success, or an error code < 0 on failure.

int **uv_udp_getsockname** (const *uv_udp_t* handle*, struct sockaddr* *name*, int* *namelen*)

Get the local IP and port of the UDP handle.

Parameters

- **handle** – UDP handle. Should have been initialized with `uv_udp_init()` and bound.
- **name** – Pointer to the structure to be filled with the address data. In order to support IPv4 and IPv6 *struct sockaddr_storage* should be used.
- **namelen** – On input it indicates the data of the *name* field. On output it indicates how much of it was filled.

Returns 0 on success, or an error code < 0 on failure.

int **uv_udp_set_membership** (*uv_udp_t* handle*, const char* *multicast_addr*, const char* *interface_addr*, *uv_membership membership*)

Set membership for a multicast address

Parameters

- **handle** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **multicast_addr** – Multicast address to set membership for.
- **interface_addr** – Interface address.
- **membership** – Should be `UV_JOIN_GROUP` or `UV_LEAVE_GROUP`.

Returns 0 on success, or an error code < 0 on failure.

int **uv_udp_set_multicast_loop** (*uv_udp_t* handle*, int *on*)

Set IP multicast loop flag. Makes multicast packets loop back to local sockets.

Parameters

- **handle** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **on** – 1 for on, 0 for off.

Returns 0 on success, or an error code < 0 on failure.

int **`uv_udp_set_multicast_ttl`**(*uv_udp_t* handle*, int *ttl*)
Set the multicast ttl.

Parameters

- **`handle`** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **`ttl`** – 1 through 255.

Returns 0 on success, or an error code < 0 on failure.

int **`uv_udp_set_multicast_interface`**(*uv_udp_t* handle*, const char* *interface_addr*)
Set the multicast interface to send or receive data on.

Parameters

- **`handle`** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **`interface_addr`** – interface address.

Returns 0 on success, or an error code < 0 on failure.

int **`uv_udp_set_broadcast`**(*uv_udp_t* handle*, int *on*)
Set broadcast on or off.

Parameters

- **`handle`** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **`on`** – 1 for on, 0 for off.

Returns 0 on success, or an error code < 0 on failure.

int **`uv_udp_set_ttl`**(*uv_udp_t* handle*, int *ttl*)
Set the time to live.

Parameters

- **`handle`** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **`ttl`** – 1 through 255.

Returns 0 on success, or an error code < 0 on failure.

int **`uv_udp_send`**(*uv_udp_send_t* req*, *uv_udp_t* handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, const struct sockaddr* *addr*, *uv_udp_send_cb send_cb*)
Send data over the UDP socket. If the socket has not previously been bound with `uv_udp_bind()` it will be bound to 0.0.0.0 (the "all interfaces" IPv4 address) and a random port number.

Parameters

- **`req`** – UDP request handle. Need not be initialized.
- **`handle`** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **`bufs`** – List of buffers to send.
- **`nbufs`** – Number of buffers in *bufs*.
- **`addr`** – *struct sockaddr_in* or *struct sockaddr_in6* with the address and port of the remote peer.
- **`send_cb`** – Callback to invoke when the data has been sent out.

Returns 0 on success, or an error code < 0 on failure.

int **uv_udp_try_send**(*uv_udp_t* handle*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, const struct sockaddr* *addr*)

Same as `uv_udp_send()`, but won't queue a send request if it can't be completed immediately.

Returns >= 0: number of bytes sent (it matches the given buffer size). < 0: negative error code (`UV_EAGAIN` is returned when the message can't be sent immediately).

int **uv_udp_recv_start**(*uv_udp_t* handle*, *uv_alloc_cb alloc_cb*, *uv_udp_recv_cb recv_cb*)

Prepare for receiving data. If the socket has not previously been bound with `uv_udp_bind()` it is bound to 0.0.0.0 (the "all interfaces" IPv4 address) and a random port number.

Parameters

- **handle** – UDP handle. Should have been initialized with `uv_udp_init()`.
- **alloc_cb** – Callback to invoke when temporary storage is needed.
- **recv_cb** – Callback to invoke with received data.

Returns 0 on success, or an error code < 0 on failure.

int **uv_udp_recv_stop**(*uv_udp_t* handle*)

Stop listening for incoming datagrams.

Parameters

- **handle** – UDP handle. Should have been initialized with `uv_udp_init()`.

Returns 0 on success, or an error code < 0 on failure.

See also:

The `uv_handle_t` API functions also apply.

6.20 uv_fs_event_t — FS Event handle

FS Event handles allow the user to monitor a given path for changes, for example, if the file was renamed or there was a generic change in it. This handle uses the best backend for the job on each platform.

6.20.1 Data types

uv_fs_event_t

FS Event handle type.

void **(*uv_fs_event_cb)**(*uv_fs_event_t* handle*, const char* *filename*, int *events*, int *status*)

Callback passed to `uv_fs_event_start()` which will be called repeatedly after the handle is started. If the handle was started with a directory the *filename* parameter will be a relative path to a file contained in the directory. The *events* parameter is an ORed mask of `uv_fs_event` elements.

uv_fs_event

Event types that `uv_fs_event_t` handles monitor.

```
enum uv_fs_event {
    UV_RENAME = 1,
    UV_CHANGE = 2
};
```

uv_fs_event_flags

Flags that can be passed to `uv_fs_event_start()` to control its behavior.

```
enum uv_fs_event_flags {
    /*
    * By default, if the fs event watcher is given a directory name, we will
    * watch for all events in that directory. This flags overrides this behavior
    * and makes fs_event report only changes to the directory entry itself. This
    * flag does not affect individual files watched.
    * This flag is currently not implemented yet on any backend.
    */
    UV_FS_EVENT_WATCH_ENTRY = 1,
    /*
    * By default uv_fs_event will try to use a kernel interface such as inotify
    * or kqueue to detect events. This may not work on remote filesystems such
    * as NFS mounts. This flag makes fs_event fall back to calling stat() on a
    * regular interval.
    * This flag is currently not implemented yet on any backend.
    */
    UV_FS_EVENT_STAT = 2,
    /*
    * By default, event watcher, when watching directory, is not registering
    * (is ignoring) changes in it's subdirectories.
    * This flag will override this behaviour on platforms that support it.
    */
    UV_FS_EVENT_RECURSIVE = 4
};
```

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.20.2 API

int **`uv_fs_event_init`**(*uv_loop_t* loop*, *uv_fs_event_t* handle*)
    Initialize the handle.

int **`uv_fs_event_start`**(*uv_fs_event_t* handle*, *uv_fs_event_cb cb*, const char* *path*, unsigned int *flags*)
    Start the handle with the given callback, which will watch the specified *path* for changes. *flags* can be an ORed mask of `uv_fs_event_flags`.

Note: Currently the only supported flag is `UV_FS_EVENT_RECURSIVE` and only on OSX and Windows.

int **`uv_fs_event_stop`**(*uv_fs_event_t* handle*)
    Stop the handle, the callback will no longer be called.

int **`uv_fs_event_getpath`**(*uv_fs_event_t* handle*, char* *buffer*, size_t* *size*)
    Get the path being monitored by the handle. The buffer must be preallocated by the user. Returns 0 on success or an error code < 0 in case of failure. On success, *buffer* will contain the path and *size* its length. If the buffer is not big enough UV_ENOBUFS will be returned and len will be set to the required size.

    Changed in version 1.3.0: the returned length no longer includes the terminating null byte, and the buffer is not null terminated.

See also:

The `uv_handle_t` API functions also apply.

6.21 `uv_fs_poll_t` — FS Poll handle

FS Poll handles allow the user to monitor a given path for changes. Unlike `uv_fs_event_t`, fs poll handles use *stat* to detect when a file has changed so they can work on file systems where fs event handles can't.

6.21.1 Data types

`uv_fs_poll_t`

FS Poll handle type.

void **`(*uv_fs_poll_cb)`** (*uv_fs_poll_t* handle*, int *status*, const *uv_stat_t* prev*, const *uv_stat_t* curr*)

Callback passed to `uv_fs_poll_start()` which will be called repeatedly after the handle is started, when any change happens to the monitored path.

The callback is invoked with *status < 0* if *path* does not exist or is inaccessible. The watcher is *not* stopped but your callback is not called again until something changes (e.g. when the file is created or the error reason changes).

When *status == 0*, the callback receives pointers to the old and new `uv_stat_t` structs. They are valid for the duration of the callback only.

Public members

N/A

See also:

The `uv_handle_t` members also apply.

6.21.2 API

int **`uv_fs_poll_init`** (*uv_loop_t* loop*, *uv_fs_poll_t* handle*)

Initialize the handle.

int **`uv_fs_poll_start`** (*uv_fs_poll_t* handle*, *uv_fs_poll_cb poll_cb*, const char* *path*, unsigned int *interval*)

Check the file at *path* for changes every *interval* milliseconds.

Note: For maximum portability, use multi-second intervals. Sub-second intervals will not detect all changes on many file systems.

int **`uv_fs_poll_stop`** (*uv_fs_poll_t* handle*)

Stop the handle, the callback will no longer be called.

int **`uv_fs_poll_getpath`** (*uv_fs_poll_t* handle*, char* *buffer*, size_t* *size*)

Get the path being monitored by the handle. The buffer must be preallocated by the user. Returns 0 on success or an error code < 0 in case of failure. On success, *buffer* will contain the path and *size* its length. If the buffer is not big enough UV_ENOBUFS will be returned and len will be set to the required size.

Changed in version 1.3.0: the returned length no longer includes the terminating null byte, and the buffer is not null terminated.

See also:

The *uv_handle_t* API functions also apply.

6.22 Filesystem operations

libuv provides a wide variety of cross-platform sync and async filesystem operations. All functions defined in this document take a callback, which is allowed to be NULL. If the callback is NULL the request is completed synchronously, otherwise it will be performed asynchronously.

All file operations are run on the threadpool, see *Thread pool work scheduling* for information on the threadpool size.

6.22.1 Data types

uv_fs_t

Filesystem request type.

uv_timespec_t

Portable equivalent of `struct timespec`.

```
typedef struct {
    long tv_sec;
    long tv_nsec;
} uv_timespec_t;
```

uv_stat_t

Portable equivalent of `struct stat`.

```
typedef struct {
    uint64_t st_dev;
    uint64_t st_mode;
    uint64_t st_nlink;
    uint64_t st_uid;
    uint64_t st_gid;
    uint64_t st_rdev;
    uint64_t st_ino;
    uint64_t st_size;
    uint64_t st_blksize;
    uint64_t st_blocks;
    uint64_t st_flags;
    uint64_t st_gen;
    uv_timespec_t st_atim;
    uv_timespec_t st_mtim;
    uv_timespec_t st_ctim;
    uv_timespec_t st_birthtim;
} uv_stat_t;
```

uv_fs_type

Filesystem request type.

```
typedef enum {
    UV_FS_UNKNOWN = -1,
    UV_FS_CUSTOM,
    UV_FS_OPEN,
    UV_FS_CLOSE,
    UV_FS_READ,
    UV_FS_WRITE,
    UV_FS_SENDFILE,
    UV_FS_STAT,
    UV_FS_LSTAT,
    UV_FS_FSTAT,
    UV_FS_FTRUNCATE,
    UV_FS_UTIME,
    UV_FS_FUTIME,
    UV_FS_ACCESS,
    UV_FS_CHMOD,
    UV_FS_FCHMOD,
    UV_FS_FSYNC,
    UV_FS_FDATASYNC,
    UV_FS_UNLINK,
    UV_FS_RMDIR,
    UV_FS_MKDIR,
    UV_FS_MKDTEMP,
    UV_FS_RENAME,
    UV_FS_SCANDIR,
    UV_FS_LINK,
    UV_FS_SYMLINK,
    UV_FS_READLINK,
    UV_FS_CHOWN,
    UV_FS_FCHOWN
} uv_fs_type;
```

`uv_dirent_t`
Cross platform (reduced) equivalent of `struct dirent`. Used in *`uv_fs_scandir_next()`*.

```
    typedef enum {
        UV_DIRENT_UNKNOWN,
        UV_DIRENT_FILE,
        UV_DIRENT_DIR,
        UV_DIRENT_LINK,
        UV_DIRENT_FIFO,
        UV_DIRENT_SOCKET,
        UV_DIRENT_CHAR,
        UV_DIRENT_BLOCK
    } uv_dirent_type_t;

    typedef struct uv_dirent_s {
        const char* name;
        uv_dirent_type_t type;
    } uv_dirent_t;
```

Public members

*uv_loop_t** **`uv_fs_t.loop`**
Loop that started this request and where completion will be reported. Readonly.

uv_fs_type **`uv_fs_t.fs_type`**
FS request type.

const char* **`uv_fs_t.path`**
Path affecting the request.

ssize_t **`uv_fs_t.result`**

Result of the request. < 0 means error, success otherwise. On requests such as `uv_fs_read()` or `uv_fs_write()` it indicates the amount of data that was read or written, respectively.

uv_stat_t **`uv_fs_t.statbuf`**

Stores the result of `uv_fs_stat()` and other stat requests.

void* **`uv_fs_t.ptr`**

Stores the result of `uv_fs_readlink()` and serves as an alias to *statbuf*.

See also:

The `uv_req_t` members also apply.

6.22.2 API

void **`uv_fs_req_cleanup`**(*uv_fs_t* req*)

Cleanup request. Must be called after a request is finished to deallocate any memory libuv might have allocated.

int **`uv_fs_close`**(*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, uv_fs_cb *cb*)

Equivalent to close(2).

int **`uv_fs_open`**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, int *flags*, int *mode*, uv_fs_cb *cb*)

Equivalent to open(2).

Note: On Windows libuv uses *CreateFileW* and thus the file is always opened in binary mode. Because of this the O_BINARY and O_TEXT flags are not supported.

int **`uv_fs_read`**(*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, int64_t *offset*, uv_fs_cb *cb*)

Equivalent to preadv(2).

int **`uv_fs_unlink`**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)

Equivalent to unlink(2).

int **`uv_fs_write`**(*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, const *uv_buf_t bufs[]*, unsigned int *nbufs*, int64_t *offset*, uv_fs_cb *cb*)

Equivalent to pwritev(2).

int **`uv_fs_mkdir`**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, int *mode*, uv_fs_cb *cb*)

Equivalent to mkdir(2).

Note: *mode* is currently not implemented on Windows.

int **`uv_fs_mkdtemp`**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *tpl*, uv_fs_cb *cb*)

Equivalent to mkdtemp(3).

Note: The result can be found as a null terminated string at *req->path*.

int **`uv_fs_rmdir`**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)

Equivalent to rmdir(2).

int **`uv_fs_scandir`**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, int *flags*, uv_fs_cb *cb*)

int **uv_fs_scandir_next** (*uv_fs_t* req*, *uv_dirent_t* ent*)
Equivalent to scandir(3), with a slightly different API. Once the callback for the request is called, the user can use `uv_fs_scandir_next()` to get *ent* populated with the next directory entry data. When there are no more entries `UV_EOF` will be returned.

Note: Unlike *scandir(3)*, this function does not return the "." and ".." entries.

Note: On Linux, getting the type of an entry is only supported by some filesystems (btrfs, ext2, ext3 and ext4 at the time of this writing), check the getdents(2) man page.

int **uv_fs_stat** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)

int **uv_fs_fstat** (*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, uv_fs_cb *cb*)

int **uv_fs_lstat** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)
Equivalent to stat(2), fstat(2) and fstat(2) respectively.

int **uv_fs_rename** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, const char* *new_path*, uv_fs_cb *cb*)
Equivalent to rename(2).

int **uv_fs_fsync** (*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, uv_fs_cb *cb*)
Equivalent to fsync(2).

int **uv_fs_fdatasync** (*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, uv_fs_cb *cb*)
Equivalent to fdatasync(2).

int **uv_fs_ftruncate** (*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, int64_t *offset*, uv_fs_cb *cb*)
Equivalent to ftruncate(2).

int **uv_fs_sendfile** (*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file out_fd*, *uv_file in_fd*, int64_t *in_offset*, size_t *length*, uv_fs_cb *cb*)
Limited equivalent to sendfile(2).

int **uv_fs_access** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, int *mode*, uv_fs_cb *cb*)
Equivalent to access(2) on Unix. Windows uses `GetFileAttributesW()`.

int **uv_fs_chmod** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, int *mode*, uv_fs_cb *cb*)

int **uv_fs_fchmod** (*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, int *mode*, uv_fs_cb *cb*)
Equivalent to chmod(2) and fchmod(2) respectively.

int **uv_fs_utime** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, double *atime*, double *mtime*, uv_fs_cb *cb*)

int **uv_fs_futime** (*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, double *atime*, double *mtime*, uv_fs_cb *cb*)
Equivalent to utime(2) and futime(2) respectively.

int **uv_fs_link** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, const char* *new_path*, uv_fs_cb *cb*)
Equivalent to link(2).

int **uv_fs_symlink** (*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, const char* *new_path*, int *flags*, uv_fs_cb *cb*)
Equivalent to symlink(2).

Note: On Windows the *flags* parameter can be specified to control how the symlink will be created:

- `UV_FS_SYMLINK_DIR`: indicates that *path* points to a directory.
- `UV_FS_SYMLINK_JUNCTION`: request that the symlink is created using junction points.

int **uv_fs_readlink**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)
Equivalent to readlink(2).

int **uv_fs_realpath**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_fs_cb *cb*)
Equivalent to realpath(3) on Unix. Windows uses `GetFinalPathNameByHandle()`.

Note: This function is not implemented on Windows XP and Windows Server 2003. On these systems, UV_ENOSYS is returned.

New in version 1.8.0.

int **uv_fs_chown**(*uv_loop_t* loop*, *uv_fs_t* req*, const char* *path*, uv_uid_t *uid*, uv_gid_t *gid*, uv_fs_cb *cb*)

int **uv_fs_fchown**(*uv_loop_t* loop*, *uv_fs_t* req*, *uv_file file*, uv_uid_t *uid*, uv_gid_t *gid*, uv_fs_cb *cb*)
Equivalent to chown(2) and fchown(2) respectively.

Note: These functions are not implemented on Windows.

See also:

The `uv_req_t` API functions also apply.

6.23 Thread pool work scheduling

libuv provides a threadpool which can be used to run user code and get notified in the loop thread. This thread pool is internally used to run all filesystem operations, as well as getaddrinfo and getnameinfo requests.

Its default size is 4, but it can be changed at startup time by setting the `UV_THREADPOOL_SIZE` environment variable to any value (the absolute maximum is 128).

The threadpool is global and shared across all event loops. When a particular function makes use of the threadpool (i.e. when using `uv_queue_work()`) libuv preallocates and initializes the maximum number of threads allowed by `UV_THREADPOOL_SIZE`. This causes a relatively minor memory overhead (~1MB for 128 threads) but increases the performance of threading at runtime.

Note: Note that even though a global thread pool which is shared across all events loops is used, the functions are not thread safe.

6.23.1 Data types

`uv_work_t`
Work request type.

void **(*uv_work_cb)**(*uv_work_t* req*)
Callback passed to `uv_queue_work()` which will be run on the thread pool.

void **(*uv_after_work_cb)**(*uv_work_t* req*, int *status*)
Callback passed to `uv_queue_work()` which will be called on the loop thread after the work on the threadpool has been completed. If the work was cancelled using `uv_cancel()` *status* will be `UV_ECANCELED`.

Public members

*uv_loop_t** **`uv_work_t.loop`**

Loop that started this request and where completion will be reported. Readonly.

See also:

The `uv_req_t` members also apply.

6.23.2 API

int **`uv_queue_work`**(*uv_loop_t* loop*, *uv_work_t* req*, *uv_work_cb work_cb*, *uv_after_work_cb after_work_cb*)

Initializes a work request which will run the given *work_cb* in a thread from the threadpool. Once *work_cb* is completed, *after_work_cb* will be called on the loop thread.

This request can be cancelled with `uv_cancel()`.

See also:

The `uv_req_t` API functions also apply.

6.24 DNS utility functions

libuv provides asynchronous variants of *getaddrinfo* and *getnameinfo*.

6.24.1 Data types

`uv_getaddrinfo_t`

getaddrinfo request type.

void **`(*uv_getaddrinfo_cb)`**(*uv_getaddrinfo_t* req*, int *status*, struct addrinfo* *res*)

Callback which will be called with the getaddrinfo request result once complete. In case it was cancelled, *status* will have a value of `UV_ECANCELED`.

`uv_getnameinfo_t`

getnameinfo request type.

void **`(*uv_getnameinfo_cb)`**(*uv_getnameinfo_t* req*, int *status*, const char* *hostname*, const char* *service*)

Callback which will be called with the getnameinfo request result once complete. In case it was cancelled, *status* will have a value of `UV_ECANCELED`.

Public members

*uv_loop_t** **`uv_getaddrinfo_t.loop`**

Loop that started this getaddrinfo request and where completion will be reported. Readonly.

struct addrinfo* **`uv_getaddrinfo_t.addrinfo`**

Pointer to a *struct addrinfo* containing the result. Must be freed by the user with `uv_freeaddrinfo()`.

Changed in version 1.3.0: the field is declared as public.

*uv_loop_t** **`uv_getnameinfo_t.loop`**

Loop that started this getnameinfo request and where completion will be reported. Readonly.

char[NI_MAXHOST] **uv_getnameinfo_t.host**

Char array containing the resulting host. It's null terminated.

Changed in version 1.3.0: the field is declared as public.

char[NI_MAXSERV] **uv_getnameinfo_t.service**

Char array containing the resulting service. It's null terminated.

Changed in version 1.3.0: the field is declared as public.

See also:

The `uv_req_t` members also apply.

6.24.2 API

int **uv_getaddrinfo**(*uv_loop_t* loop*, *uv_getaddrinfo_t* req*, *uv_getaddrinfo_cb getaddrinfo_cb*, const char* *node*, const char* *service*, const struct addrinfo* *hints*)

Asynchronous getaddrinfo(3).

Either node or service may be NULL but not both.

hints is a pointer to a struct addrinfo with additional address type constraints, or NULL. Consult *man -s 3 getaddrinfo* for more details.

Returns 0 on success or an error code < 0 on failure. If successful, the callback will get called sometime in the future with the lookup result, which is either:

- status == 0, the res argument points to a valid *struct addrinfo*, or
- status < 0, the res argument is NULL. See the UV_EAI_* constants.

Call `uv_freeaddrinfo()` to free the addrinfo structure.

Changed in version 1.3.0: the callback parameter is now allowed to be NULL, in which case the request will run **synchronously**.

void **uv_freeaddrinfo**(struct addrinfo* *ai*)

Free the struct addrinfo. Passing NULL is allowed and is a no-op.

int **uv_getnameinfo**(*uv_loop_t* loop*, *uv_getnameinfo_t* req*, *uv_getnameinfo_cb getnameinfo_cb*, const struct sockaddr* *addr*, int *flags*)

Asynchronous getnameinfo(3).

Returns 0 on success or an error code < 0 on failure. If successful, the callback will get called sometime in the future with the lookup result. Consult *man -s 3 getnameinfo* for more details.

Changed in version 1.3.0: the callback parameter is now allowed to be NULL, in which case the request will run **synchronously**.

See also:

The `uv_req_t` API functions also apply.

6.25 Shared library handling

libuv provides cross platform utilities for loading shared libraries and retrieving symbols from them, using the following API.

6.25.1 Data types

`uv_lib_t`
  Shared library data type.

Public members

N/A

6.25.2 API

int **`uv_dlopen`**(const char* *filename*, *uv_lib_t** *lib*)
  Opens a shared library. The filename is in utf-8. Returns 0 on success and -1 on error. Call `uv_dlerror()` to get the error message.

void **`uv_dlclose`**(*uv_lib_t** *lib*)
  Close the shared library.

int **`uv_dlsym`**(*uv_lib_t** *lib*, const char* *name*, void** *ptr*)
  Retrieves a data pointer from a dynamic library. It is legal for a symbol to map to NULL. Returns 0 on success and -1 if the symbol was not found.

const char* **`uv_dlerror`**(const *uv_lib_t** *lib*)
  Returns the last uv_dlopen() or uv_dlsym() error message.

6.26 Threading and synchronization utilities

libuv provides cross-platform implementations for multiple threading and synchronization primitives. The API largely follows the pthreads API.

6.26.1 Data types

`uv_thread_t`
  Thread data type.

void **`(*uv_thread_cb)`**(void* *arg*)
  Callback that is invoked to initialize thread execution. *arg* is the same value that was passed to `uv_thread_create()`.

`uv_key_t`
  Thread-local key data type.

`uv_once_t`
  Once-only initializer data type.

`uv_mutex_t`
  Mutex data type.

`uv_rwlock_t`
  Read-write lock data type.

`uv_sem_t`
  Semaphore data type.

uv_cond_t
Condition data type.

uv_barrier_t
Barrier data type.

6.26.2 API

Threads

int **uv_thread_create** (*uv_thread_t* tid*, *uv_thread_cb entry*, void* *arg*)
Changed in version 1.4.1: returns a UV_E* error code on failure

uv_thread_t **uv_thread_self** (void)

int **uv_thread_join** (*uv_thread_t *tid*)

int **uv_thread_equal** (const *uv_thread_t* t1*, const *uv_thread_t* t2*)

Thread-local storage

Note: The total thread-local storage size may be limited. That is, it may not be possible to create many TLS keys.

int **uv_key_create** (*uv_key_t* key*)

void **uv_key_delete** (*uv_key_t* key*)

void* **uv_key_get** (*uv_key_t* key*)

void **uv_key_set** (*uv_key_t* key*, void* *value*)

Once-only initialization

Runs a function once and only once. Concurrent calls to `uv_once()` with the same guard will block all callers except one (it's unspecified which one). The guard should be initialized statically with the UV_ONCE_INIT macro.

void **uv_once** (*uv_once_t* guard*, void (**callback*)(void))

Mutex locks

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

int **uv_mutex_init** (*uv_mutex_t* handle*)

void **uv_mutex_destroy** (*uv_mutex_t* handle*)

void **uv_mutex_lock** (*uv_mutex_t* handle*)

int **uv_mutex_trylock** (*uv_mutex_t* handle*)

void **uv_mutex_unlock** (*uv_mutex_t* handle*)

Read-write locks

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

int **uv_rwlock_init** (*uv_rwlock_t* rwlock*)

void **uv_rwlock_destroy** (*uv_rwlock_t* rwlock*)

void **uv_rwlock_rdlock** (*uv_rwlock_t* rwlock*)

int **uv_rwlock_tryrdlock** (*uv_rwlock_t* rwlock*)

void **uv_rwlock_rdunlock** (*uv_rwlock_t* rwlock*)

void **uv_rwlock_wrlock** (*uv_rwlock_t* rwlock*)

int **uv_rwlock_trywrlock** (*uv_rwlock_t* rwlock*)

void **uv_rwlock_wrunlock** (*uv_rwlock_t* rwlock*)

Semaphores

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

int **uv_sem_init** (*uv_sem_t* sem*, unsigned int *value*)

void **uv_sem_destroy** (*uv_sem_t* sem*)

void **uv_sem_post** (*uv_sem_t* sem*)

void **uv_sem_wait** (*uv_sem_t* sem*)

int **uv_sem_trywait** (*uv_sem_t* sem*)

Conditions

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

Note: Callers should be prepared to deal with spurious wakeups on `uv_cond_wait()` and `uv_cond_timedwait()`.

int **uv_cond_init** (*uv_cond_t* cond*)

void **uv_cond_destroy** (*uv_cond_t* cond*)

void **uv_cond_signal** (*uv_cond_t* cond*)

void **uv_cond_broadcast** (*uv_cond_t* cond*)

void **uv_cond_wait** (*uv_cond_t* cond*, *uv_mutex_t* mutex*)

int **uv_cond_timedwait** (*uv_cond_t* cond*, *uv_mutex_t* mutex*, uint64_t *timeout*)

Barriers

Functions return 0 on success or an error code < 0 (unless the return type is void, of course).

Note: `uv_barrier_wait()` returns a value > 0 to an arbitrarily chosen "serializer" thread to facilitate cleanup, i.e.

```
if (uv_barrier_wait(&barrier) > 0)
    uv_barrier_destroy(&barrier);
```

int **uv_barrier_init** (*uv_barrier_t* barrier*, unsigned int *count*)

void **uv_barrier_destroy** (*uv_barrier_t* barrier*)

int **uv_barrier_wait** (*uv_barrier_t* barrier*)

6.27 Miscellaneous utilities

This section contains miscellaneous functions that don't really belong in any other section.

6.27.1 Data types

uv_buf_t
Buffer data type.

char* **uv_buf_t.base**
Pointer to the base of the buffer. Readonly.

size_t **uv_buf_t.len**
Total bytes in the buffer. Readonly.

Note: On Windows this field is ULONG.

void* **(*uv_malloc_func)** (size_t *size*)
Replacement function for malloc(3). See `uv_replace_allocator()`.

void* **(*uv_realloc_func)** (void* *ptr*, size_t *size*)
Replacement function for realloc(3). See `uv_replace_allocator()`.

void* **(*uv_calloc_func)** (size_t *count*, size_t *size*)
Replacement function for calloc(3). See `uv_replace_allocator()`.

void **(*uv_free_func)** (void* *ptr*)
Replacement function for free(3). See `uv_replace_allocator()`.

uv_file
Cross platform representation of a file handle.

uv_os_sock_t
Cross platform representation of a socket handle.

uv_os_fd_t
Abstract representation of a file descriptor. On Unix systems this is a *typedef* of *int* and on Windows a *HANDLE*.

uv_rusage_t
Data type for resource usage results.

```
    typedef struct {
        uv_timeval_t ru_utime; /* user CPU time used */
        uv_timeval_t ru_stime; /* system CPU time used */
        uint64_t ru_maxrss; /* maximum resident set size */
        uint64_t ru_ixrss; /* integral shared memory size */
        uint64_t ru_idrss; /* integral unshared data size */
        uint64_t ru_isrss; /* integral unshared stack size */
        uint64_t ru_minflt; /* page reclaims (soft page faults) */
        uint64_t ru_majflt; /* page faults (hard page faults) */
        uint64_t ru_nswap; /* swaps */
        uint64_t ru_inblock; /* block input operations */
        uint64_t ru_oublock; /* block output operations */
        uint64_t ru_msgsnd; /* IPC messages sent */
        uint64_t ru_msgrcv; /* IPC messages received */
        uint64_t ru_nsignals; /* signals received */
        uint64_t ru_nvcsw; /* voluntary context switches */
        uint64_t ru_nivcsw; /* involuntary context switches */
    } uv_rusage_t;
```

uv_cpu_info_t

Data type for CPU information.

```
    typedef struct uv_cpu_info_s {
        char* model;
        int speed;
        struct uv_cpu_times_s {
            uint64_t user;
            uint64_t nice;
            uint64_t sys;
            uint64_t idle;
            uint64_t irq;
        } cpu_times;
    } uv_cpu_info_t;
```

uv_interface_address_t

Data type for interface addresses.

```
    typedef struct uv_interface_address_s {
        char* name;
        char phys_addr[6];
        int is_internal;
        union {
            struct sockaddr_in address4;
            struct sockaddr_in6 address6;
        } address;
        union {
            struct sockaddr_in netmask4;
            struct sockaddr_in6 netmask6;
        } netmask;
    } uv_interface_address_t;
```

uv_passwd_t

Data type for password file information.

```
    typedef struct uv_passwd_s {
        char* username;
        long uid;
        long gid;
        char* shell;
        char* homedir;
    } uv_passwd_t;
```

6.27.2 API

uv_handle_type **uv_guess_handle** (*uv_file file*)

Used to detect what type of stream should be used with a given file descriptor. Usually this will be used during initialization to guess the type of the stdio streams.

For isatty(3) equivalent functionality use this function and test for `UV_TTY`.

int **uv_replace_allocator** (*uv_malloc_func malloc_func*, *uv_realloc_func realloc_func*, *uv_calloc_func calloc_func*, *uv_free_func free_func*)

New in version 1.6.0.

Override the use of the standard library's malloc(3), calloc(3), realloc(3), free(3), memory allocation functions.

This function must be called before any other libuv function is called or after all resources have been freed and thus libuv doesn't reference any allocated memory chunk.

On success, it returns 0, if any of the function pointers is NULL it returns UV_EINVAL.

Warning: There is no protection against changing the allocator multiple times. If the user changes it they are responsible for making sure the allocator is changed while no memory was allocated with the previous allocator, or that they are compatible.

uv_buf_t **uv_buf_init** (char* *base*, unsigned int *len*)

Constructor for `uv_buf_t`.

Due to platform differences the user cannot rely on the ordering of the *base* and *len* members of the uv_buf_t struct. The user is responsible for freeing *base* after the uv_buf_t is done. Return struct passed by value.

char** **uv_setup_args** (int *argc*, char** *argv*)

Store the program arguments. Required for getting / setting the process title.

int **uv_get_process_title** (char* *buffer*, size_t *size*)

Gets the title of the current process.

int **uv_set_process_title** (const char* *title*)

Sets the current process title.

int **uv_resident_set_memory** (size_t* *rss*)

Gets the resident set size (RSS) for the current process.

int **uv_uptime** (double* *uptime*)

Gets the current system uptime.

int **uv_getrusage** (*uv_rusage_t** *rusage*)

Gets the resource usage measures for the current process.

Note: On Windows not all fields are set, the unsupported fields are filled with zeroes.

int **uv_cpu_info** (*uv_cpu_info_t*** *cpu_infos*, int* *count*)

Gets information about the CPUs on the system. The *cpu_infos* array will have *count* elements and needs to be freed with `uv_free_cpu_info()`.

void **uv_free_cpu_info** (*uv_cpu_info_t** *cpu_infos*, int *count*)

Frees the *cpu_infos* array previously allocated with `uv_cpu_info()`.

int **uv_interface_addresses** (*uv_interface_address_t*** *addresses*, int* *count*)

Gets address information about the network interfaces on the system. An array of *count* elements is allocated and returned in *addresses*. It must be freed by the user, calling `uv_free_interface_addresses()`.

void **uv_free_interface_addresses** (*uv_interface_address_t* addresses*, int *count*)
Free an array of `uv_interface_address_t` which was returned by `uv_interface_addresses()`.

void **uv_loadavg** (double *avg[3]*)
Gets the load average. See: http://en.wikipedia.org/wiki/Load_(computing)

Note: Returns [0,0,0] on Windows (i.e., it's not implemented).

int **uv_ip4_addr** (const char* *ip*, int *port*, struct sockaddr_in* *addr*)
Convert a string containing an IPv4 addresses to a binary structure.

int **uv_ip6_addr** (const char* *ip*, int *port*, struct sockaddr_in6* *addr*)
Convert a string containing an IPv6 addresses to a binary structure.

int **uv_ip4_name** (const struct sockaddr_in* *src*, char* *dst*, size_t *size*)
Convert a binary structure containing an IPv4 address to a string.

int **uv_ip6_name** (const struct sockaddr_in6* *src*, char* *dst*, size_t *size*)
Convert a binary structure containing an IPv6 address to a string.

int **uv_inet_ntop** (int *af*, const void* *src*, char* *dst*, size_t *size*)

int **uv_inet_pton** (int *af*, const char* *src*, void* *dst*)
Cross-platform IPv6-capable implementation of inet_ntop(3) and inet_pton(3). On success they return 0. In case of error the target *dst* pointer is unmodified.

int **uv_exepath** (char* *buffer*, size_t* *size*)
Gets the executable path.

int **uv_cwd** (char* *buffer*, size_t* *size*)
Gets the current working directory.

Changed in version 1.1.0: On Unix the path no longer ends in a slash.

int **uv_chdir** (const char* *dir*)
Changes the current working directory.

int **uv_os_homedir** (char* *buffer*, size_t* *size*)
Gets the current user's home directory. On Windows, *uv_os_homedir()* first checks the *USERPROFILE* environment variable using *GetEnvironmentVariableW()*. If *USERPROFILE* is not set, *GetUserProfileDirectoryW()* is called. On all other operating systems, *uv_os_homedir()* first checks the *HOME* environment variable using getenv(3). If *HOME* is not set, getpwuid_r(3) is called. The user's home directory is stored in *buffer*. When *uv_os_homedir()* is called, *size* indicates the maximum size of *buffer*. On success *size* is set to the string length of *buffer*. On *UV_ENOBUFS* failure *size* is set to the required length for *buffer*, including the null byte.

Warning: *uv_os_homedir()* is not thread safe.

New in version 1.6.0.

int **uv_os_tmpdir** (char* *buffer*, size_t* *size*)
Gets the temp directory. On Windows, *uv_os_tmpdir()* uses *GetTempPathW()*. On all other operating systems, *uv_os_tmpdir()* uses the first environment variable found in the ordered list *TMPDIR*, *TMP*, *TEMP*, and *TEMPDIR*. If none of these are found, the path *"/tmp"* is used, or, on Android, *"/data/local/tmp"* is used. The temp directory is stored in *buffer*. When *uv_os_tmpdir()* is called, *size* indicates the maximum size of *buffer*. On success *size* is set to the string length of *buffer* (which does not include the terminating null). On *UV_ENOBUFS* failure *size* is set to the required length for *buffer*, including the null byte.

Warning: *uv_os_tmpdir()* is not thread safe.

New in version 1.9.0.

int **uv_os_get_passwd**(*uv_passwd_t** *pwd*)

Gets a subset of the password file entry for the current effective uid (not the real uid). The populated data includes the username, euid, gid, shell, and home directory. On non-Windows systems, all data comes from getpwuid_r(3). On Windows, uid and gid are set to -1 and have no meaning, and shell is *NULL*. After successfully calling this function, the memory allocated to *pwd* needs to be freed with `uv_os_free_passwd()`.

New in version 1.9.0.

void **uv_os_free_passwd**(*uv_passwd_t** *pwd*)

Frees the *pwd* memory previously allocated with `uv_os_get_passwd()`.

New in version 1.9.0.

uint64_t **uv_get_total_memory**(void)

Gets memory information (in bytes).

uint64_t **uv_hrtime**(void)

Returns the current high-resolution real time. This is expressed in nanoseconds. It is relative to an arbitrary time in the past. It is not related to the time of day and therefore not subject to clock drift. The primary use is for measuring performance between intervals.

Note: Not every platform can support nanosecond resolution; however, this value will always be in nanoseconds.

void **uv_print_all_handles**(*uv_loop_t** *loop*, FILE* *stream*)

Prints all handles associated with the given *loop* to the given *stream*.

Example:

```
uv_print_all_handles(uv_default_loop(), stderr);
/*
[--I] signal   0x1a25ea8
[-AI] async    0x1a25cf0
[R--] idle     0x1a7a8c8
*/
```

The format is *[flags] handle-type handle-address*. For *flags*:

- *R* is printed for a handle that is referenced
- *A* is printed for a handle that is active
- *I* is printed for a handle that is internal

Warning: This function is meant for ad hoc debugging, there is no API/ABI stability guarantees.

New in version 1.8.0.

void **uv_print_active_handles**(*uv_loop_t** *loop*, FILE* *stream*)

This is the same as `uv_print_all_handles()` except only active handles are printed.

Warning: This function is meant for ad hoc debugging, there is no API/ABI stability guarantees.

New in version 1.8.0.