
Python libuv CFFI Bindings

Release 0.1.0.dev0

October 10, 2016

1	Contents:	1
1.1	Errors – exceptions and error handling	1
1.2	Loop – event loop	11
1.3	Handle – handle base classes	16
1.4	Async – async handle	19
1.5	Check – check handle	20
1.6	Idle – idle handle	21
1.7	Pipe – pipe handle	21
1.8	Poll – poll handle	23
1.9	Prepare – poll handle	25
1.10	Process – process handle	26
1.11	Signal – signal handle	28
1.12	Timer – timer handle	29
1.13	Stream – stream handle	31
1.14	TCP – TCP handle	36
1.15	TTY – TTY handle	38
1.16	UDP – UDP handle	39
1.17	FSEvent – fs event handle	43
1.18	FSPoll – fs poll handle	45
1.19	Request – request base classes	47
1.20	DNS utilities	48
1.20.1	Requests	48
1.20.2	Data Structures	50
1.20.3	Enumerations	51
2	Indices and tables	53
	Python Module Index	55

Contents:

1.1 Errors – exceptions and error handling

class `uv.error.StatusCodes`

Status codes enumeration. Status codes are instances of this class and — beside SUCCESS — vary across platforms. Status codes other than SUCCESS are linked with a corresponding exception.

SUCCESS = None

Success — no error occurred.

Type `uv.StatusCodes`

E2BIG = None

Argument list too long.

Type `uv.StatusCodes`

EACCES = None

Permission denied.

Type `uv.StatusCodes`

EADDRINUSE = None

Address already in use.

Type `uv.StatusCodes`

EADDRNOTAVAIL = None

Address not available.

Type `uv.StatusCodes`

EAFNOSUPPORT = None

Address family not supported.

Type `uv.StatusCodes`

EAGAIN = None

Resource temporarily unavailable.

Type `uv.StatusCodes`

EAI_ADDRFAMILY = None

Address family not supported.

Type `uv.StatusCodes`

EAI_AGAIN = None
Temporary failure.
Type uv.StatusCodes

EAI_BADFLAGS = None
Bad address flags value.
Type uv.StatusCodes

EAI_BADHINTS = None
Invalid value for hints.
Type uv.StatusCodes

EAI_CANCELED = None
Request canceled.
Type uv.StatusCodes

EAI_FAIL = None
Permanent failure.
Type uv.StatusCodes

EAI_FAMILY = None
Address family not supported.
Type uv.StatusCodes

EAI_MEMORY = None
Out of memory.
Type uv.StatusCodes

EAI_NODATA = None
No address.
Type uv.StatusCodes

EAI_NONAME = None
Unknown node or service.
Type uv.StatusCodes

EAI_OVERFLOW = None
Argument buffer overflow.
Type uv.StatusCodes

EAI_PROTOCOL = None
Resolved protocol is unknown.
Type uv.StatusCodes

EAI_SERVICE = None
Service not available for socket type.
Type uv.StatusCodes

EAI_SOCKTYPE = None
Socket type not supported.
Type uv.StatusCodes

EALREADY = None
Connection already in progress.

Type uv.StatusCodes

EBADF = None

Bad file descriptor.

Type uv.StatusCodes

EBUSY = None

Resource busy or locked.

Type uv.StatusCodes

ECANCELED = None

Operation canceled.

Type uv.StatusCodes

ECHARSET = None

Invalid Unicode character.

Type uv.StatusCodes

ECONNABORTED = None

Software caused connection abort.

Type uv.StatusCodes

ECONNREFUSED = None

Connection refused.

Type uv.StatusCodes

ECONNRESET = None

Connection reset by peer.

Type uv.StatusCodes

EDESTADDRREQ = None

Destination address required.

Type uv.StatusCodes

EEXIST = None

File already exists.

Type uv.StatusCodes

EFAULT = None

Bad address in system call argument.

Type uv.StatusCodes

EFBIG = None

File too large.

Type uv.StatusCodes

EHOSTUNREACH = None

Host is unreachable.

Type uv.StatusCodes

EINTR = None

Interrupted system call.

Type uv.StatusCodes

EINVAL = None

Invalid argument.

Type uv.StatusCodes

EIO = None

IO error.

Type uv.StatusCodes

EISCONN = None

Socket is already connected.

Type uv.StatusCodes

EISDIR = None

Illegal operation on a directory.

Type uv.StatusCodes

ELOOP = None

Too many symbolic links encountered.

Type uv.StatusCodes

EMFILE = None

Too many open files.

Type uv.StatusCodes

EMSGSIZE = None

Message too long.

Type uv.StatusCodes

ENAMETOOLONG = None

Name too long.

Type uv.StatusCodes

ENETDOWN = None

Network is down.

Type uv.StatusCodes

ENETUNREACH = None

Network is unreachable.

Type uv.StatusCodes

ENFILE = None

File table overflow.

Type uv.StatusCodes

ENOBUFS = None

No buffer space available.

Type uv.StatusCodes

ENODEV = None

No such device.

Type uv.StatusCodes

ENOENT = None

No such file or directory.

Type uv.StatusCodes

ENOMEM = None

Not enough memory.

Type uv.StatusCodes

ENONET = None

Machine is not on the network.

Type uv.StatusCodes

ENOPROTOPT = None

Protocol not available.

Type uv.StatusCodes

ENOSPC = None

No space left on device.

Type uv.StatusCodes

ENOSYS = None

Function not implemented.

Type uv.StatusCodes

ENOTCONN = None

Socket is not connected.

Type uv.StatusCodes

ENOTDIR = None

Not a directory.

Type uv.StatusCodes

ENOTEMPTY = None

Directory not empty.

Type uv.StatusCodes

ENOTSOCK = None

Socket operation on non-socket.

Type uv.StatusCodes

ENOTSUP = None

Operation not supported on socket.

Type uv.StatusCodes

EPERM = None

Operation not permitted.

Type uv.StatusCodes

EPIPE = None

Broken pipe.

Type uv.StatusCodes

EPROTO = None

Protocol error.

Type uv.StatusCodes

EPROTONOSUPPORT = None

Protocol not supported.

Type uv.StatusCodes

EPROTOTYPE = None

Protocol wrong type for socket.

Type uv.StatusCodes

ERANGE = None

Result too large.

Type uv.StatusCodes

EROFS = None

Read-only file system.

Type uv.StatusCodes

ESHUTDOWN = None

Cannot send after transport endpoint shutdown.

Type uv.StatusCodes

ESPIPE = None

Invalid seek.

Type uv.StatusCodes

ESRCH = None

No such process.

Type uv.StatusCodes

ETIMEDOUT = None

Connection timed out.

Type uv.StatusCodes

ETXTBSY = None

Text file is busy.

Type uv.StatusCodes

EXDEV = None

Cross-device link not permitted.

Type uv.StatusCodes

UNKNOWN = None

Unknown error.

Type uv.StatusCodes

EOF = None

End of file.

Type uv.StatusCodes

ENXIO = None

No such device or address.

Type uv.StatusCodes

EMLINK = None

Too many links.

Type uv.StatusCodes

EHOSTDOWN = None

Host is down.

Type uv.StatusCodes

exception

Corresponding exception (subclass of `uv.error.UVError`).

Readonly True

Return type Subclass[uv.error.UVError]

message

Human readable error message.

Readonly True

Return type unicode

classmethod `get` (*code*)

Look up the given status code and return the corresponding instance of `uv.StatusCodes` or the original integer if there is no such status code.

Parameters `code` (`uv.StatusCodes` | `int` | `None`) – potential status code

Returns status code instance or original status code integer

Return type uv.StatusCodes | int

exception `uv.error.UVError` (*code=None, message=''*)

Base class of all uv-related exceptions.

code = None

Error-Code

Readonly True

Type uv.StatusCodes | int | None

name = None

Error-Name

Readonly True

Type unicode

message = None

Error-Message

Readonly True

Type unicode

exception `uv.error.ArgumentError` (*code=None, message=''*)

Invalid arguments.

exception `uv.error.TemporaryUnavailableError` (*code=None, message=''*)

Resource temporary unavailable.

exception `uv.error.CanceledError` (*code=None, message=''*)

Request canceled.

exception `uv.error.PermanentError` (*code=None, message=''*)

Permanent failure.

exception `uv.error.PermissionError` (*code=None, message=''*)
Permission denied.

exception `uv.error.BadFileDescriptorError` (*code=None, message=''*)
Bad file descriptor.

exception `uv.error.ResourceBusyError` (*code=None, message=''*)
Resource busy or locked.

exception `uv.error.CharsetError` (*code=None, message=''*)
Invalid unicode character.

exception `uv.error.FileExistsError` (*code=None, message=''*)
File already exists.

exception `uv.error.FileTooLargeError` (*code=None, message=''*)
File too large.

exception `uv.error.InterruptedError` (*code=None, message=''*)
Interrupted system call.

exception `uv.error.IOException` (*code=None, message=''*)
Generic IO related error.

exception `uv.error.IsConnectedError` (*code=None, message=''*)
Socket is already connected.

exception `uv.error.IsADirectoryError` (*code=None, message=''*)
Illegal operation on a directory.

exception `uv.error.NotADirectoryError` (*code=None, message=''*)
Not a directory.

exception `uv.error.NotEmptyError` (*code=None, message=''*)
Directory is not empty.

exception `uv.error.MessageTooLongError` (*code=None, message=''*)
Message too long.

exception `uv.error.NameTooLongError` (*code=None, message=''*)
Name too long.

exception `uv.error.BufferSpaceError` (*code=None, message=''*)
No buffer space available.

exception `uv.error.NoSpaceError` (*code=None, message=''*)
No space left on the device.

exception `uv.error.NotImplementedError` (*code=None, message=''*)
Function not implemented.

exception `uv.error.NotConnectedError` (*code=None, message=''*)
Socket is not connected.

exception `uv.error.HostUnreachableError` (*code=None, message=''*)
Host is unreachable.

exception `uv.error.ResultTooLargeError` (*code=None, message=''*)
Result too large.

exception `uv.error.SeekError` (*code=None, message=''*)
Invalid seek.

exception `uv.error.ProcessLookupError` (*code=None, message=''*)
No such progress.

exception `uv.error.TimeoutError` (*code=None, message=''*)
Operation timed out.

exception `uv.error.CrossDeviceError` (*code=None, message=''*)
Cross device link not permitted.

exception `uv.error.EOFError` (*code=None, message=''*)
End of file error.

exception `uv.error.UnsupportedOperation` (*code=None, message=''*)
Base class of all unsupported operation related errors.

exception `uv.error.ClosedStructureError`
Invalid operation on closed structure.

exception `uv.error.ClosedHandleError`
Invalid operation on closed handle.

exception `uv.error.ClosedLoopError`
Invalid operation on closed loop.

exception `uv.error.NotSocketError` (*code=None, message=''*)
Socket operation on non-socket.

exception `uv.error.NotSupportedError` (*code=None, message=''*)
Operation not supported on socket.

exception `uv.error.ProtocolError` (*code=None, message=''*)
Protocol error.

exception `uv.error.ProtocolNoOptionError` (*code=None, message=''*)
Protocol option unavailable.

exception `uv.error.ProtocolNotSupportedError` (*code=None, message=''*)
Protocol not supported.

exception `uv.error.ProtocolTypeError` (*code=None, message=''*)
Protocol wrong type for socket.

exception `uv.error.AddressError` (*code=None, message=''*)
Base class of all address related errors.

exception `uv.error.AddressUnavailableError` (*code=None, message=''*)
Address not available.

exception `uv.error.AddressInUseError` (*code=None, message=''*)
Address already in use.

exception `uv.error.AddressFamilyError` (*code=None, message=''*)
Address family not supported.

exception `uv.error.AddressFlagsError` (*code=None, message=''*)
Bad address flags value.

exception `uv.error.AddressHintsError` (*code=None, message=''*)
Bad address hints value.

exception `uv.error.AddressDataError` (*code=None, message=''*)
No address given.

exception `uv.error.AddressNameError` (*code=None, message=''*)
Unknown node or service.

exception `uv.error.AddressProtocolError` (*code=None, message=''*)
Resolved protocol is unknown.

exception `uv.error.AddressServiceError` (*code=None, message=''*)
Service not available for socket type.

exception `uv.error.AddressSocketTypeError` (*code=None, message=''*)
Socket type not supported.

exception `uv.error.DestinationAddressError` (*code=None, message=''*)
Destination address required.

exception `uv.error.ConnectionError` (*code=None, message=''*)
Base class of all connection related errors.

exception `uv.error.BrokenPipeError` (*code=None, message=''*)
Broken pipe.

exception `uv.error.ConnectionAbortedError` (*code=None, message=''*)
Software caused connection abort.

exception `uv.error.ConnectionRefusedError` (*code=None, message=''*)
Connection refused.

exception `uv.error.ConnectionResetError` (*code=None, message=''*)
Connection reset by peer.

exception `uv.error.ConnectionInProgressError` (*code=None, message=''*)
Connection already in progress.

exception `uv.error.NotFoundError` (*code=None, message=''*)
Base class of all not found related errors.

exception `uv.error.DeviceNotFoundError` (*code=None, message=''*)
No such device or address.

exception `uv.error.FileNotFoundError` (*code=None, message=''*)
No such file or directory.

exception `uv.error.NetworkError` (*code=None, message=''*)
Base class of all network related errors.

exception `uv.error.NetworkDownError` (*code=None, message=''*)
Network is down.

exception `uv.error.NetworkUnreachableError` (*code=None, message=''*)
Network is unreachable.

exception `uv.error.NoNetworkError` (*code=None, message=''*)
Machine is not on the network.

exception `uv.error.SystemFailureError` (*code=None, message=''*)
Base class of all system related errors.

exception `uv.error.MemoryError` (*code=None, message=''*)
Not enough memory.

exception `uv.error.TooManyLinksError` (*code=None, message=''*)
Too many links encountered.

exception `uv.error.TooManySymbolicLinksError` (*code=None, message=''*)
Too many symbolic links encountered.

exception `uv.error.TooManyOpenFilesError` (*code=None, message=''*)
Too many open files.

exception `uv.error.FileTableOverflowError` (*code=None, message=''*)
File table overflow.

1.2 Loop – event loop

class `uv.Loop` (*allocator=None, buffer_size=65536, default=False*)

The event loop is the central part of this library. It takes care of polling for IO and scheduling callbacks to be run based on different sources of events.

Raises

- **RuntimeError** – error while initializing global default loop
- **UVError** – error initializing the new event loop

Parameters

- **allocator** (`uv.loop.Allocator`) – read buffer allocator
- **buffer_size** (*int*) – size of the default allocators read buffer
- **default** (*bool*) – instantiate the default loop

classmethod `get_default` (*instantiate=True, **keywords*)

Get the default (across multiple threads) event loop. Note that although this returns the same loop across multiple threads loops are not thread safe. Normally there is one thread running the default loop and others interfering with it trough `uv.Async` handles or `uv.Loop.call_later()`.

Parameters **instantiate** (*bool*) – instantiate the default event loop if it does not exist

Returns global default loop

Return type *Loop*

classmethod `get_current` (*instantiate=True, **keywords*)

Get the current (thread local) default event loop. Loops register themselves as current loop on instantiation and in their `uv.Loop.run()` method.

Parameters **instantiate** (*bool*) – instantiate a new loop if there is no current loop

Returns current thread's default loop

Return type *Loop*

excepthook = None

If an exception occurs during the execution of a callback this excepthook is called with the corresponding event loop and exception details. The default behavior is to print the traceback to stderr and stop the event loop. To override the default behavior assign a custom function to this attribute.

Note: If the excepthook raises an exception itself the program would be in an undefined state. Therefore it terminates with `sys.exit(1)` in that case immediately.

excepthook (*loop, exc_type, exc_value, exc_traceback*)

Parameters

- **loop** (`uv.Loop`) – corresponding event loop
- **exc_type** (`type`) – exception type (subclass of `BaseException`)
- **exc_value** (`BaseException`) – exception instance
- **exc_traceback** (`traceback`) – traceback which encapsulates the call stack at the point where the exception originally occurred

Readonly False

Type Callable[[`uv.Loop`, `type`, `Exception`, `traceback.Traceback`], `None`]

exc_type = None

Type of last exception handled by the excepthook.

Readonly True

Type `type`

exc_value = None

Instance of last exception handled by the excepthook.

Readonly True

Type `BaseException`

exc_traceback = None

Traceback of the last exception handled by the excepthook.

Readonly True

Type `traceback`

closed

True if and only if the loop has been closed.

Readonly True

Return type `bool`

alive

True if there are active and referenced handles running on the loop, False otherwise.

Readonly True

Return type `bool`

now

Current internal timestamp in milliseconds. The timestamp increases monotonically from some arbitrary point in time.

Readonly True

Return type `int`

handles

Set of all handles running on the loop.

Readonly True

Return type `set`

fileno ()

Get the file descriptor of the backend. This is only supported on kqueue, epoll and event ports.

Raises

- **uv.UVError** – error getting file descriptor

- **uv.ClosedLoopError** – loop has already been closed

Returns backend file descriptor

Return type `int`

make_current ()

Make the loop the current thread local default loop.

update_time ()

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.

Raises **uv.ClosedLoopError** – loop has already been closed

Note: 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.

get_timeout ()

Get the poll timeout. The return value is in milliseconds, or -1 for no timeout.

Raises **uv.ClosedLoopError** – loop has already been closed

Returns backend timeout in milliseconds

Return type `int`

run (*mode*=<*RunModes.DEFAULT: 0*>)

Run the loop in the specified mode.

Raises **uv.ClosedLoopError** – loop has already been closed

Parameters *mode* (`uv.RunModes`) – run mode

Returns run mode specific return value

Return type `bool`

stop ()

Stop the event loop, causing `uv.Loop.run()` to end as soon as possible. This will happen not sooner than the next loop iteration. If this method was called before blocking for IO, the loop will not block for IO on this iteration.

close ()

Closes all internal loop resources. This method must only be called once the loop has finished its execution or it will raise `uv.error.ResourceBusyError`.

Note: Loops are automatically closed when they are garbage collected. However because the exact time this happens is non-deterministic you should close them explicitly.

Raises

- **uv.UVError** – error while closing the loop
- **uv.error.ResourceBusyError** – loop is currently running or there are pending operations

close_all_handles (*on_closed*=None)

Close all handles.

Parameters on_closed (*Callable*[[*uv.Handle*], *None*]) – callback which should run after a handle has been closed (overrides the current callback if specified)

call_later (*callback*, **arguments*, ***keywords*)

Schedule a callback to run at some later point in time. The callback does not keep the loop alive if there are no other active handles running on the loop.

This method is thread safe.

Parameters

- **callback** (*callable*) – callback which should run at some later point in time
- **arguments** (*tuple*) – arguments that should be passed to the callback
- **keywords** (*dict*) – keyword arguments that should be passed to the callback

reset_exception ()

Reset the last exception caught by the excepthook.

on_wakeup ()

Called after the event loop has been woken up.

Warning: This method is only for internal purposes and is not part of the official API. You should never call it directly!

handle_exception ()

Handle the current exception using the excepthook.

Warning: This method is only for internal purposes and is not part of the official API. You should never call it directly!

structure_set_pending (*structure*)

Add a structure to the set of pending structures.

Warning: This method is only for internal purposes and is not part of the official API. You should never call it directly!

structure_clear_pending (*structure*)

Remove a structure from the set of pending structures.

Warning: This method is only for internal purposes and is not part of the official API. You should never call it directly!

structure_is_pending (*structure*)

Return true if and only if the structure is pending.

Warning: This method is only for internal purposes and is not part of the official API. You should never call it directly!

class uv.RunModes

Run modes to control the behavior of *uv.Loop.run()*.

DEFAULT = None

Run the event loop until there are no more active and referenced handles or requests. *uv.Loop.run()*

returns *True* if `uv.Loop.stop()` was called and there are still active handles or requests and *False* otherwise.

Type `uv.RunModes`

ONCE = None

Poll for IO once. Note that `uv.Loop.run()` will block if there are no pending callbacks. `uv.Loop.run()` returns *True* if there are still active handles or requests which means the event loop should run again sometime in the future.

Type `uv.RunModes`

NOWAIT = None

Poll for IO once but do not block if there are no pending callbacks. `uv.Loop.run()` returns *True* if there are still active handles or requests which means the event loop should run again sometime in the future.

Type `uv.RunModes`

class `uv.loop.Allocator`

Abstract base class for read buffer allocators. Allows swappable allocation strategies and custom read result types.

Warning: This class exposes some details of the underlying CFFI based wrapper — use it with caution. Any errors in the allocator might lead to unpredictable behavior.

allocate (*handle, suggested_size, uv_buffer*)

Called if libuv needs a new read buffer. The allocated chunk of memory has to be assigned to `uv_buf.base` and the length of the chunk to `uv_buf.len` use `library.uv_buffer_set()` for assigning. Base might be *NULL* which triggers an *ENOBUFS* error in the read callback.

Parameters

- **handle** (`uv.Handle`) – handle caused the read
- **suggested_size** (*int*) – suggested buffer size
- **uv_buffer** (`ffi.CData[uv_buf_t]`) – uv target buffer

finalize (*handle, length, uv_buffer*)

Called in the read callback to access the read buffer's data. The result of this call is directly passed to the user's read callback which allows to use a custom read result type.

Parameters

- **handle** (`uv.Handle`) – handle caused the read
- **length** (*int*) – length of bytes read
- **uv_buffer** (`ffi.CData[uv_buf_t]`) – uv buffer used for reading

Returns buffer's data (default type is `bytes`)

Return type `Any | bytes`

class `uv.loop.DefaultAllocator` (*buffer_size=65536*)

Default read buffer allocator which only uses one buffer and copies the data to a python `bytes` object after reading.

1.3 Handle – handle base classes

class `uv.Handle`

Handles represent long-lived objects capable of performing certain operations while active. This is the abstract base class of all internal libuv and pure Python handles.

Note: Handles underlie a special garbage collection strategy which means they are not garbage collected as other objects. If a handle is able to do anything in the program for example calling a callback they are not garbage collected.

`loop`

Loop the handle is running on.

Readonly True

Return type `uv.Loop`

`closing`

Handle is already closed or is closing. This is `True` right after `close` has been called. Operations on a closed or closing handle will raise `uv.ClosedHandleError`.

Readonly True

Return type `bool`

`closed`

Handle has been closed. This is `True` right after the `close` callback has been called. It means all internal resources are freed and this handle is ready to be garbage collected.

Readonly True

Return type `bool`

`active`

Handle is active or not. What “active” means depends on the handle type:

`uv.Async`: is always active and cannot be deactivated

`uv.Pipe`, `uv.TCP`, `uv.UDP`, ...: basically any handle dealing with IO is active when it is doing something involves IO like reading, writing, connecting or listening

`uv.Check`, `uv.Idle`, `uv.Timer`, ...: handle is active when it has been started and not yet stopped

Readonly True

Type `bool`

`referenced`

Handle is referenced or not. If the event loop runs in default mode it will exit when there are no more active and referenced handles left. This has nothing to do with CPython’s reference counting.

Readonly False

Type `bool`

`reference()`

Reference the handle. If the event loop runs in default mode it will exit when there are no more active and referenced handles left. This has nothing to do with CPython’s reference counting. References are idempotent, that is, if a handle is referenced calling this method again will have not effect.

Raises `uv.ClosedHandleError` – handle has already been closed or is closing

dereference ()

Dereference the handle. If the event loop runs in default mode it will exit when there are no more active and referenced handles left. This has nothing to do with CPython's reference counting. References are idempotent, that is, if a handle is not referenced calling this method again will have no effect.

Raises `uv.ClosedHandleError` – handle has already been closed or is closing

close (*on_closed=None*)

Close the handle. Please make sure to call this method on any handle you do not need anymore. This method is idempotent, that is, if the handle is already closed or is closing calling it will have no effect at all.

In-progress requests, like connect or write requests, are cancelled and have their callbacks called asynchronously with `uv.StatusCodes.ECANCELED`.

After this method has been called on a handle no operations can be performed on it (they raise `uv.ClosedHandleError`).

Note: Handles are automatically closed when they are garbage collected. However because the exact time this happens is non-deterministic you should close all handles explicitly. Especially if they handle external resources.

Parameters `on_closed` (*Callable[[uv.Handle], None]*) – callback which should run after the handle has been closed (overrides the current callback if specified)

class `uv.UVHandle` (*loop, arguments=()*)

Base class of all internal libuv based handles.

Raises `uv.LoopClosedError` – loop has already been closed

Parameters

- `loop` (`uv.Loop`) – loop where the handle should run on
- `arguments` (*tuple*) – arguments passed to the libuv handle init function

on_closed

Callback which should run after the handle has been closed.

on_closed (*handle*)

Parameters `handle` (`uv.Handle`) – handle which has been closed

Readonly False

Type `Callable[[uv.Handle], None]`

data

User-specific data of any type. This is necessary because of the usage of slots.

Readonly False

Type Any

allocator

Allocator used to allocate new read buffers for this handle.

Readonly False

Type `uv.loop.Allocator`

closing

Readonly True

Type bool

closed

Readonly True

Type bool

active

Readonly True

Type bool

referenced

Readonly False

Type bool

send_buffer_size

Size of the send buffer that the operating system uses for the socket. The following handles are supported: TCP and UDP handles on Unix and Windows, Pipe handles only on Unix. On all unsupported handles this will raise `uv.UVError` with error code `EINVAL` (`uv.error.ArgumentError`).

Note: Unlike libuv this library abstracts the different behaviours on Linux and other operating systems. This means, the size set is divided by two on Linux because Linux internally multiplies it by two.

Raises

- `uv.UVError` – error while getting/setting the send buffer size
- `uv.ClosedHandleError` – handle has already been closed or is closing

Readonly False

Type int

receive_buffer_size

Size of the receive buffer that the operating system uses for the socket. The following handles are supported: TCP and UDP handles on Unix and Windows, Pipe handles only on Unix. On all unsupported handles this will raise `uv.UVError` with error code `EINVAL` (`uv.error.ArgumentError`).

Note: Unlike libuv this library abstracts the different behaviours on Linux and other operating systems. This means, the size set is divided by two on Linux because Linux internally multiplies it by two.

Raises

- `uv.UVError` – error while getting/setting the receive buffer size
- `uv.ClosedHandleError` – handle has already been closed or is closing

Readonly False

Type int

fileno()

Get the platform dependent file descriptor equivalent. The following handles are supported: TCP, UDP, TTY, Pipes and Poll. On all other handles this will raise `uv.UVError` with error code `EINVAL` (`uv.error.ArgumentError`).

If a handle does not have an attached file descriptor yet this method will raise `uv.UVError` with error code `EBADF` (`uv.error.BadFileDescriptorError`).

Warning: Be very careful when using this method. Libuv assumes it is in control of the file descriptor so any change to it may result in unpredictable malfunctions.

Raises

- `uv.UVError` – error while receiving fileno
- `uv.ClosedHandleError` – handle has already been closed or is closing

Returns platform dependent file descriptor equivalent

Return type `int`

reference()

Raises `uv.ClosedHandleError` – handle has already been closed or is closing

dereference()

Raises `uv.ClosedHandleError` – handle has already been closed or is closing

close (*on_closed=None*)

Parameters `on_closed` (`Callable[[uv.Handle], None]`) – callback which should run after the handle has been closed (overrides the current callback if specified)

set_pending()

Warning: This method is only for internal purposes and is not part of the official API. It deactivates the garbage collection for the handle which means the handle and the corresponding loop are excluded from garbage collection. You should never call it directly!

clear_pending()

Warning: This method is only for internal purposes and is not part of the official API. It reactivates the garbage collection for the handle. You should never call it directly!

1.4 Async – async handle

class `uv.Async` (*loop=None, on_wakeup=None*)

Async handles are able to wakeup the event loop of another thread and run the given callback in the event loop's thread. Although the `uv.Async.send()` method is thread-safe the constructor is not.

To run a callback in the event loop's thread without creating an `uv.Async` handle use `uv.Loop.call_later()`.

Raises `uv.UVError` – error while initializing the handle

Parameters

- `loop` (`uv.Loop`) – event loop the handle should run on

- **on_wakeup** (*Callable*[[*uv.Async*], *None*]) – callback which should run in the event loop’s thread after the event loop has been woken up

on_wakeup

Callback which should run in the event loop’s thread after the event loop has been woken up.

on_wakeup (*async_handle*)

Parameters **async_handle** (*uv.Async*) – handle the call originates from

Readonly False

Type *Callable*[[*uv.Async*], *None*]

send (*on_wakeup=None*)

Wakeup the event loop and run the callback afterwards. Multiple calls to this method are coalesced if they happen before the callback has been called. This means not every call will yield an execution of the callback. It is safe to call this method from outside the event loop’s thread.

Raises

- **uv.UVError** – error while trying to wakeup the event loop
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **on_wakeup** (*Callable*[[*uv.Async*], *None*]) – callback which should run in the event loop’s thread after the event loop has been woken up (overrides the current callback if specified)

1.5 Check – check handle

class *uv.Check* (*loop=None, on_check=None*)

Check handles will run the given callback once per loop iteration, right after polling for IO after they have been started.

Raises **uv.UVError** – error while initializing the handle

Parameters

- **loop** (*uv.Loop*) – event loop the handle should run on
- **on_check** (*Callable*[[*uv.Check*], *None*]) – callback which should run right after polling for IO after the handle has been started

on_check

Callback which should run right after polling for IO after the handle has been started.

on_check (*check_handle*)

Parameters **check_handle** (*uv.Check*) – handle the call originates from

Readonly False

Type *Callable*[[*uv.Check*], *None*]

start (*on_check=None*)

Start the handle. The callback will be called once per loop iteration right after polling for IO from now on.

Raises

- **uv.UVError** – error while starting the handle
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters `on_check` (*Callable*[[*uv.Check*], *None*]) – callback which should run right after polling for IO (overrides the current callback if specified)

stop()

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

Raises `uv.UVError` – error while stopping the handle

1.6 Idle – idle handle

class `uv.Idle` (*loop=None, on_idle=None*)

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

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 IO.

Raises `uv.UVError` – error while initializing the handle

Parameters

- `loop` (*uv.Loop*) – event loop the handle should run on
- `on_idle` (*Callable*[[*uv.Idle*], *None*]) – callback which should run right before the prepare handles after the handle has been started

on_idle

Callback which should run right before the prepare handles after the handle has been started.

on_idle (*idle*)

Parameters `idle` (*uv.Idle*) – handle the call originates from

Readonly `False`

Type `Callable`[[*uv.Idle*], *None*]

start (*on_idle=None*)

Start the handle. The callback will run once per loop iteration right before the prepare handles from now on.

Raises

- `uv.UVError` – error while starting the handle
- `uv.HandleClosedError` – handle has already been closed or is closing

Parameters `on_idle` (*Callable*[[*uv.Idle*], *None*]) – callback which should run right before the prepare handles (overrides the current callback if specified)

stop()

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

Raises `uv.UVError` – error while stopping the handle

1.7 Pipe – pipe handle

class `uv.Pipe` (*ipc=False, loop=None, on_read=None, on_connection=None*)

Stream interface to local domain sockets on Unix and named pipes on Windows, which supports inter process communication.

raises **uv.UVError** error while initializing the handle

Parameters

- **ipc** (*bool*) – pipe should have inter process communication support not
- **loop** (*uv.Loop*) – event loop the handle should run on
- **on_read** (*Callable[[uv.Pipe, uv.StatusCodes, bytes], None]:type on_connection: Callable[[uv.Pipe, uv.StatusCodes, bytes], None]*) – callback which should be called when data has been read
- **on_connection** – callback which should run after a new connection has been made or on error (if stream is in listen mode)

open (*fd*)

Open an existing file descriptor as a pipe handle.

Raises

- **uv.UVError** – error while opening the file descriptor
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **fd** (*int*) – file descriptor

bind (*path*)

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

Raises

- **uv.UVError** – error while binding to *path*
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **path** (*unicode*) – path or name to bind to to bind to

connect (*path, on_connect=None*)

Connect to the given Unix domain socket or named pipe.

Raises **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **path** (*unicode*) – path to connect to
- **on_connect** (*Callable[[uv.PipeConnectRequest, uv.StatusCode], None]:rtype: uv.PipeConnectRequest*) – callback which should run after a connection has been established or on error

pending_count

Number of pending streams to receive over IPC.

Readonly True

Return type *int*

pending_type

Type of first pending stream, if there is a pending stream. Returns a subclass of *uv.UVStream*.

Readonly True

Return type *type | None*

pending_accept (**arguments, **keywords*)

Accept a pending stream.

Raises

- **uv.UVError** – error while accepting stream
- **uv.ClosedHandleError** – handle has already been closed or is closing

Return type *uv.UVStream*

pending_instances (*amount*)

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

Note: This setting applies to Windows only.

Raises **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **amount** (*int*) – amount of pending instances

sockname

Name of the Unix domain socket or the named pipe.

Raises

- **uv.UVError** – error while receiving sockname
- **uv.ClosedHandleError** – handle has already been closed or is closing

Readonly True

Return type unicode

peername

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

Raises

- **uv.UVError** – error while receiving peername
- **uv.ClosedHandleError** – handle has already been closed or is closing

Readonly True

Return type unicode

class **uv.PipeConnectRequest** (*pipe, path, on_connect=None*)

Pipe specific connect request.

Parameters

- **pipe** (*uv.Pipe*) – pipe to establish a connection on
- **path** (*unicode*) – path to connect to
- **on_connect** (*Callable[[uv.PipeConnectRequest, uv.StatusCode], None]*) – callback which should run after a connection has been established or on error

1.8 Poll – poll handle

class **uv.Poll** (*fd, loop=None, on_event=None*)

Poll handles are used to watch file descriptors for readability and writability. The purpose of poll handles is to

enable integrating external libraries that rely on the event loop to signal them about the socket status changes. Using them for any other purpose is not recommended. Use `uv.TCP`, `uv.UDP`, etc. instead, which provide faster and more scalable implementations, than what can be archived with `uv.Poll`, especially on Windows.

It is possible that poll handles occasionally signal that a file descriptor is readable or writable even when it is not. 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.

Do 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 `uv.Poll.stop()` or `uv.Handle.close()` has been called.

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

Raises `uv.UVError` – error while initializing the handle

Parameters

- **fd** (*int*) – file descriptor to be polled (is set to non-blocking mode)
- **loop** (`uv.Loop`) – event loop the handle should run on
- **on_event** (*Callable*[[`uv.Poll`, `uv.StatusCode`, *int*], *None*]) – callback which should be called on IO events after the handle has been started

fd

File descriptor the handle polls on.

Readonly `True`

Type `int`

on_event

Callback which should be called on IO events after the handle has been started.

on_event (*poll_handle*, *status*, *events*)

Parameters

- **poll_handle** (`uv.Poll`) – handle the call originates from
- **status** (`uv.StatusCode`) – may indicate any errors
- **events** (*int*) – bitmask of the triggered IO events

Readonly `False`

Type `Callable`[[`uv.Poll`, `uv.StatusCode`, *int*], *None*]

fileno ()

Number of the file descriptor polled on.

Return type `int`

start (*events*=`<PollEvent.READABLE: 1>`, *on_event*=*None*)

Start polling the file descriptor for the given events. As soon as an event is detected the callback will be called with status code class:`uv.StatusCode.SUCCESS` and the triggered events.

If an error happens while polling the callback gets called with status code `!= 0` which corresponds to a `uv.StatusCode`.

Calling this on a handle that is already active is fine. Doing so will update the events mask that is being polled for.

Raises

- **uv.UVError** – error while starting the handle
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **events** (*int*) – bitmask of events to be polled for
- **on_event** (*Callable[[uv.Poll, uv.StatusCode, int], None]*) – callback which should be called on IO events (overrides the current callback if specified)

stop()

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

:raises uv.UVError error while stopping the handle

class uv.PollEvent

Events reported by *uv.Poll* on IO events.

READABLE = None

File descriptor is readable.

Type uv.PollEvent

WRITABLE = None

File descriptor is writable.

Type uv.PollEvent

1.9 Prepare – poll handle

class uv.Prepare (loop=None, on_prepare=None)

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

Raises uv.UVError – error while initializing the handle

Parameters

- **loop** (*uv.Loop*) – event loop the handle should run on
- **on_prepare** (*Callable[[uv.Prepare], None]*) – callback which should run right before polling for IO after the handle has been started

on_prepare

Callback which should run right before polling for IO after the handle has been started.

on_prepare (prepare_handle)

Parameters **prepare_handle** (*uv.Prepare*) – handle the call originates from

Readonly False

Type Callable[[uv.Prepare], None]

start (on_prepare=None)

Start the handle. The callback will run once per loop iteration right before polling for IO from now on.

Raises

- **uv.UVError** – error while starting the handle
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters on `prepare` (*Callable*[[*uv.Prepare*], *None*]) – callback which should run right before polling for IO (overrides the current callback if specified)

stop()

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

Raises **uv.UVError** – error while stopping the handle

1.10 Process – process handle

class **uv.Process** (*arguments*, *uid=None*, *gid=None*, *cwd=None*, *env=None*, *stdin=None*, *stdout=None*, *stderr=None*, *stdio=None*, *flags=<ProcessFlags.WINDOWS_HIDE: 16>*, *loop=None*, *on_exit=None*)

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

Raises **uv.UVError** – error while initializing the handle

Parameters

- **arguments** (*list*[*unicode*]) – program path and command line arguments
- **uid** (*int*) – spawn as user with user id *uid*
- **gid** (*int*) – spawn as group with group id *gid*
- **cwd** (*unicode*) – child process working directory
- **env** (*dict*[*unicode*, *unicode*]) – child process environment variables
- **flags** (*int*) – process spawn flags to be used
- **stdin** (*int* | *uv.UVStream* | *uv.CreatePipe* | *file-like* | *None*) – standard input of the child process
- **stdout** (*int* | *uv.UVStream* | *uv.CreatePipe* | *file-like* | *None*) – standard output of the child process
- **stderr** (*int* | *uv.UVStream* | *uv.CreatePipe* | *file-like* | *None*) – standard error of the child process
- **stdio** (*list*[*int* | *uv.UVStream* | *uv.CreatePipe* | *file-like*]) – other standard file descriptors of the child process
- **loop** (*uv.Loop*) – event loop the handle should run on
- **on_exit** (*Callable*[[*uv.Process*, *int*, *int*], *None*]) – callback which should be called after process exited

stdin = None

Standard input of the child process.

Readonly True

Type *int* | *uv.UVStream* | *file-like* | *None*

stdout = None

Standard output of the child process.

Readonly True

Type int | uv.UVStream | file-like | None

stderr = None

Standard error of the child process.

Readonly True

Type int | uv.UVStream | file-like | None

stdio = None

Other standard file descriptors of the child process.

Readonly True

Type list[int | uv.UVStream | file-like]

on_exit = None

Callback which should be called after process exited.

on_exit (*process_handle*, *returncode*, *signum*)

Parameters

- **process_handle** (*uv.Process*) – handle the call originates from
- **returncode** (*int*) – status code returned by the process on termination
- **signum** (*int*) – signal number caused the process to exit

Readonly False

Type Callable[[uv.Process, int, int], None]

pid

PID of the spawned process.

Raises **uv.ClosedHandleError** – handle has already been closed or is closing

Readonly True

Return type int

kill (*signum*=<Signals.SIGINT: 2>)

Send the specified signal to the process.

Raises **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **signum** (*int*) – signal number

class **uv.CreatePipe** (*readable=False*, *writable=False*, *ipc=False*)

Passed to one of the standard IO arguments of *Process*, it tells the library to create a new pipe to communicate with the child process.

Parameters

- **readable** (*bool*) – pipe should be readable
- **writable** (*bool*) – pipe should be writable
- **ipc** (*bool*) – pipe should support inter process communication

uv.PIPE = <CreatePipe readable=True, writable=True, ipc=True>

Create a readable and writable inter process communication pipe.

class **uv.ProcessFlags**

Process configuration flags enumeration.

DETACHED = None

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.Handle.dereference()` on the child’s process handle.

Type `uv.ProcessFlags`

WINDOWS_HIDE = None

Hide the subprocess console window that would normally be created. This option is only meaningful on Windows systems. By default it is enabled, to disable this flag pass `0` to `uv.Process`’s flag parameter. On Unix it is ignored.

Type `uv.ProcessFlags`

WINDOWS_VERBATIM = None

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 ignored.

Type `uv.ProcessFlags`

1.11 Signal – signal handle

class `uv.Signal` (*loop=None, on_signal=None*)

Signal handles implement Unix style signal handling on a per-event loop basis. Reception of the generic `uv.Signals` is emulated on Windows. Watchers for other signals can be successfully created, but these signals are never received.

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.

Raises `uv.UVError` – error while initializing the handle

Parameters

- **loop** (`uv.Loop`) – event loop the handle should run on
- **on_signal** (`Callable[[uv.Signal, int], None]`) – callback which should be called on signal delivery after the handle has been started

on_signal

Callback which should be called on signal delivery after the handle has been started.

on_signal(`signal_handle`, `signum`):

Parameters

- **signal_handle** (`uv.Signal`) – handle the call originates from
- **signum** (`int`) – number of the received signal

Readonly `False`

Type `Callable[[uv.Signal, int], None]`

signum

Signal currently monitored by this handle.

Raises `uv.ClosedHandleError` – handle has already been closed or is closing

Readonly `True`

Return type `int`

start (*signum*, *on_signal=None*)

Start listening for the given signal.

Raises

- `uv.UVError` – error while starting the handle
- `uv.ClosedHandleError` – handle has already been closed or is closing

Parameters

- **signum** (*int*) – signal number to listen for
- **on_signal** (*Callable[[uv.Signal, int], None]*) – callback which should be called on signal delivery (overrides the current callback if specified)

stop ()

Stop listening. The callback will no longer be called.

Raises `uv.UVError` – error while stopping the handle

class `uv.Signals`

Standard cross platform signals enumeration.

SIGINT = None

Is normally delivered when the user presses CTRL+C. However it is not generated when terminal is in raw mode.

Type `uv.Signals`

SIGBREAK = None

Is delivered when the user presses CTRL+BREAK. This signal is only supported on Windows.

Type `uv.Signals`

SIGHUP = None

Is generated when the user closes the console window. After that the OS might terminate the program after a few seconds.

Type `uv.Signals`

SIGWINCH = None

Is generated when the console window has been resized. On Windows libuv emulates `SIGWINCH` when the program uses a `uv.TTY` handle to write to the console. It may not always be delivered in a timely manner, because libuv will only detect changes when the cursor is being moved. When a readable `uv.TTY` handle is used in raw mode, resizing the console buffer will also trigger `SIGWINCH`.

Type `uv.Signals`

1.12 Timer – timer handle

class `uv.Timer` (*loop=None*, *on_timeout=None*)

Timer handles are used to schedule callbacks to be called in the future after a given amount of time.

Raises `uv.UVError` – error while initializing the handle

Parameters

- **loop** (`uv.Loop`) – event loop the handle should run on
- **on_timeout** (`Callable[[uv.Timer], None]`) – callback which should run on timeout

on_timeout

Callback which should run on timeout.

on_timeout (*timer_handle*)

Parameters **timer_handle** (`uv.Timer`) – handle the call originates from

Readonly False

Type `Callable[[uv.Timer], None]`

repeat

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 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.

Raises `uv.ClosedHandleError` – handle has already been closed or is closing

Readonly False

Return type `int`

again ()

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 raises `uv.error.ArgumentError`.

Raises

- `uv.UVError` – error while restarting the timer
- `uv.ClosedHandleError` – handle has already been closed or is closing

start (*timeout*, *repeat=0*, *on_timeout=None*)

Start the timer. 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.

Raises

- `uv.UVError` – error while starting the handle
- `uv.ClosedHandleError` – handle has already been closed or is closing

:param timeout timeout to be used (in milliseconds)

Parameters

- **repeat** (*int*) – repeat interval to be used (in milliseconds)

- **on_timeout** (*Callable*[[*uv.Timer*], *None*]) – callback which should run on timeout (overrides the current callback if specified)

stop()

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

Raises `uv.UVError` – error while stopping the handle

1.13 Stream – stream handle

class `uv.Stream`

Stream handles provide a reliable ordered duplex communication channel. This is the abstract base class of all stream handles.

readable

Stream is readable or not.

Readonly `True`

Type `bool`

writable

Stream is writable or not.

Readonly `True`

Type `bool`

read_start (*on_read=**None*)

Start reading data from the stream. The read callback will be called from now on when data has been read.

Raises

- `uv.UVError` – error while start reading data from the stream
- `uv.ClosedHandleError` – handle has already been closed or is closing

Parameters *on_read* (*Callable*[[*uv.Stream*, *uv.StatusCodes*, *bytes*], *None*]) – callback which should be called when data has been read (overrides the current callback if specified)

read_stop()

Stop reading data from the stream. The read callback will no longer be called from now on.

Raises `uv.UVError` – error while stop reading data from the stream

write (*buffers*, *send_stream=**None*, *on_write=**None*)

Write data to stream. Buffers are written in the given order.

If *send_stream* is not *None* and the stream supports inter process communication this method sends *send_stream* to the other end of the connection.

Parameters

- **buffers** (*tuple*[*bytes*] | *list*[*bytes*] | *bytes*) – data which should be written
- **send_stream** (*uv.TCP* | *uv.Pipe* | *None*) – stream handle which should be send

- **on_write** (*Callable*[[*uv.Request*, *uv.StatusCodes*], *None*]:returns: *issued write request*) – callback which should run after all data has been written

Return type *uv.Request*

shutdown (*on_shutdown=None*)

Shutdown the outgoing (write) side of a duplex stream. It waits for pending write requests to complete.

Parameters **on_shutdown** (*Callable*[[*uv.ShutdownRequest*, *uv.StatusCodes*], *None*]:returns: *issued stream shutdown request*) – callback which should run after shutdown has been completed

Return type *uv.Request*

class *uv.UVStream* (*loop, ipc, arguments, on_read, on_connection*)

The base class of all libuv based streams.

Note: This class must not be instantiated directly. Please use the sub-classes for specific communication channels.

Parameters

- **loop** (*uv.Loop*) – event loop the handle should run on
- **ipc** (*bool*) – stream should support inter process communication or not
- **arguments** (*tuple*) – arguments passed to the underlying libuv initializer
- **on_read** (*Callable*[[*uv.UVStream*, *uv.StatusCodes*, *bytes*], *None*]:type *on_connection*: *Callable*[[*uv.UVStream*, *uv.StatusCodes*, *bytes*], *None*]) – callback which should be called when data has been read
- **on_connection** – callback which should run after a new connection has been made or on error (if stream is in listen mode)

on_read

Callback which should be called when data has been read.

Note: Data might be a zero-bytes long bytes object. In contrast to the Python standard library this does not indicate any error, especially not *EOF*.

on_read (*stream_handle, status, data*)

Parameters

- **stream_handle** (*uv.UVStream*) – handle the call originates from
- **status** (*uv.StatusCodes*) – status of the handle (indicate any errors)
- **data** (*bytes | Any*) – data which has been read

Readonly False

Type *Callable*[[*uv.UVStream*, *uv.StatusCodes*, *bytes*], *None*]

on_connection

Callback which should run after a new connection has been made or on error (if stream is in listen mode).

on_connection (*stream_handle, status*)

Parameters

- **stream_handle** (*uv.UVStream*) – handle the call originates from
- **status** (*uv.StatusCodes*) – status of the new connection

Readonly False**Type** Callable[[*uv.UVStream*, *uv.StatusCodes*, *uv.UVStream*], None]**ipc**

Stream does support inter process communication or not.

Readonly True**Type** bool**readable****Readonly** True**Type** bool**writable****Readonly** True**Type** bool**family**

Address family of stream, may be None.

Return type int | None**shutdown** (*on_shutdown=None*)**listen** (*backlog=5*, *on_connection=None*)

Start listening for incoming connections.

Raises

- **uv.UVError** – error while start listening for incoming connections
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **backlog** (*int*) – number of connections the kernel might queue
- **on_connection** (*Callable[[uv.UVStream, uv.StatusCodes], None]*) – callback which should run after a new connection has been made (overrides the current callback if specified)

read_start (*on_read=None*)**Raises**

- **uv.UVError** – error while start reading data from the stream
- **uv.ClosedHandleError** – handle has already been closed or is closing

read_stop ()**Raises** **uv.UVError** – error while stop reading data from the stream**write** (*buffers*, *send_stream=None*, *on_write=None*)**Return type** *uv.WriteRequest*

try_write (*buffers*)

Immediately write data to the stream without issuing a write request. Throws `uv.error.TemporaryUnavailableError` if data could not be written immediately, otherwise it returns the number of written bytes.

Raises

- `uv.UVError` – error while writing data
- `uv.ClosedHandleError` – handle has already been closed or is closing
- `uv.error.TemporaryUnavailableError` – unable to write data immediately

Parameters `buffers` (*tuple[bytes] | list[bytes] | bytes*) – data which should be written

Returns number of bytes written

Return type `int`

accept (*cls=None, *arguments, **keywords*)

Accept a new stream. This might be a new client connection or a stream sent by inter process communication.

Warning: There should be no need to use this method directly, it is mainly for internal purposes.

Raises

- `uv.UVError` – error while accepting incoming stream
- `uv.ClosedHandleError` – handle has already been closed or is closing

Parameters

- `cls` (*type*) – type of the new stream
- `arguments` (*tuple*) – arguments passed to the constructor of the new stream
- `keywords` (*dict*) – keywords passed to the constructor of the new stream

Returns new stream connection of type `cls`

Return type `uv.UVStream`

class `uv.ConnectRequest` (*stream, arguments, on_connect=None*)

Request to connect to a specific address.

Note: There is a specific connect request type for every stream type.

Parameters

- `stream` (`uv.UVStream`) – stream to establish a connection on
- `on_connect` (`Callable[[uv.ConnectRequest, uv.StatusCodes], None]`) – callback which should run after a connection has been established or on error

stream

Stream to establish a connection on.

Readonly `True`

Type `uv.UVStream`

on_connect

Callback which should run after a connection has been established.

Readonly False

Type Callable[[uv.ConnectRequest, uv.StatusCodes], None]

class uv.**WriteRequest** (*stream, buffers, send_stream=None, on_write=None*)

Request to write data to a stream and, on streams with inter process communication support, to send stream handles. Buffers are written in the given order.

Raises

- **uv.UVError** – error while initializing the request
- **uv.ClosedHandleError** – stream has already been closed or is closing

Parameters

- **stream** (uv.UVStream) – stream to write data to
- **buffers** (*tuple[bytes] | list[bytes] | bytes*) – data which should be written
- **send_stream** (uv.TCP | uv.Pipe | None) – stream handle which should be send
- **on_write** (Callable[[uv.WriteRequest, uv.StatusCodes], None]) – callback which should run after all data has been written

stream

Stream to write data to.

Readonly True

Type uv.UVStream

send_stream

Stream handle which should be send.

Readonly True

Type uv.UVStream | None

on_write

Callback which should run after all data has been written.

Readonly False

Type Callable[[uv.WriteRequest, uv.StatusCodes], None]

class uv.**ShutdownRequest** (*stream, on_shutdown=None*)

Request to shutdown the outgoing side of a duplex stream. It waits for pending write requests to complete.

Raises

- **uv.UVError** – error while initializing the request
- **uv.ClosedHandleError** – stream has already been closed or is closing

Parameters

- **stream** (uv.UVStream) – stream to shutdown
- **on_shutdown** (Callable[[uv.ShutdownRequest, uv.StatusCodes], None]) – callback which should run after shutdown has been completed

stream

Stream to shutdown.

Readonly True

Type uv.UVStream

on_shutdown

Callback which should run after shutdown has been completed.

on_shutdown (*shutdown_request, status*)

Parameters

- **shutdown_request** (*uv.ShutdownRequest*) – request the call originates from
- **status** (*uv.StatusCodes*) – status of the shutdown request

Readonly False

Type Callable[[uv.ShutdownRequest, uv.StatusCodes], None]

1.14 TCP – TCP handle

class uv.TCP (*flags=0, loop=None, on_read=None, on_connection=None*)

Stream interface to TCP sockets for clients and servers.

Raises uv.UVError – error while initializing the handle

Parameters

- **flags** (*int*) – tcp flags to be used
- **loop** (*uv.Loop*) – event loop the handle should run on
- **on_read** (*Callable[[uv.TCP, uv.StatusCodes, bytes], None]:type on_connection: Callable[[uv.TCP, uv.StatusCodes, bytes], None]*) – callback which should be called when data has been read
- **on_connection** – callback which should run after a new connection has been made or on error (if stream is in listen mode)

open (*fd*)

Open an existing file descriptor as a tcp handle.

Raises

- **uv.UVError** – error while opening the handle
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters *fd* (*int*) – file descriptor

bind (*address, flags=0*)

Bind the handle to an address. When the port is already taken, you can expect to see an `uv.StatusCode.EADDRINUSE` error from either `bind()`, `listen()` or `connect()`. That is, a successful call to this function does not guarantee that the call to `listen()` or `connect()` will succeed as well.

Raises

- **uv.UVError** – error while binding to *address*
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **address** (*uv.Address4 | uv.Address6 | tuple*) – address to bind to (*ip, port, flowinfo=0, scope_id=0*)

- **flags** (*int*) – bind flags to be used (mask of *uv.TCPFlags*)

connect (*address, on_connect=None*)

Establish an IPv4 or IPv6 TCP connection.

Raises

- **uv.UVError** – error while connecting to *address*
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **address** (*uv.Address4 | uv.Address6 | tuple*) – address to connect to
- **on_connect** (*Callable[[uv.TCPConnectRequest, uv.StatusCode], None]:rtype: uv.TCPConnectRequest*) – callback which should run after a connection has been established or on error

sockname

The current address to which the handle is bound to.

Raises

- **uv.UVError** – error while receiving sockname
- **uv.ClosedHandleError** – handle has already been closed or is closing

Readonly True

Return type *uv.Address4 | uv.Address6*

peername

The address of the peer connected to the handle.

Raises

- **uv.UVError** – error while receiving peername
- **uv.ClosedHandleError** – handle has already been closed or is closing

Readonly True

Return type *uv.Address4 | uv.Address6*

set_nodelay (*enable*)

Enable / disable Nagle's algorithm.

Raises

- **uv.UVError** – error enabling / disabling the algorithm
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **enable** (*bool*) – enable / disable

set_keepalive (*enable, delay=0*)

Enable / disable TCP keep-alive.

Raises

- **uv.UVError** – error enabling / disabling tcp keep-alive
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **enable** (*bool*) – enable / disable

- **delay** (*int*) – initial delay in seconds

set_simultaneous_accepts (*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.

Raises

- **uv.UVError** – error enabling / disabling simultaneous accepts
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **enable** (*bool*) – enable / disable

class `uv.TCPFlags`

TCP configuration enumeration.

IPV6ONLY = `None`

Disable dual stack support.

Type `uv.TCPFlags`

1.15 TTY – TTY handle

class `uv.TTY` (*fd, readable=False, loop=None, on_read=None*)

Stream interface to the local user terminal console. It allows using ANSI escape codes across platforms.

Raises **uv.UVError** – error while initializing the handle

Parameters

- **fd** (*int*) – file descriptor of the console
- **readable** (*bool*) – specifies whether the file descriptor is readable or not
- **loop** (`uv.Loop`) – event loop the handle should run on
- **on_read** (`Callable[[uv.TTY, uv.StatusCodes, bytes], None]`) – callback which should be called when data has been read

console_size

Current size of the console.

Raises **uv.UVError** – error while getting console size

Return type `ConsoleSize`

set_mode (*mode=<uv.helpers.mock.Mock object>*)

Set the the specified terminal mode.

Raises

- **uv.UVError** – error while setting mode
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **mode** (`uv.TTYMode`) – mode to set

class `uv.TTYMode`

Terminal modes enumeration.

NORMAL

Initial normal terminal mode.

Type `uv.TTYMode`

RAW

Raw input mode (on windows, `ENABLE_WINDOW_INPUT` is also enabled).

Type `uv.TTYMode`

IO

Binary-safe IO mode for IPC (Unix only).

Type `uv.TTYMode`

class `uv.ConsoleSize`

width

Width of the console.

Readonly `True`

Type `int`

height

Height of the console.

Readonly `True`

Type `int`

1.16 UDP – UDP handle

class `uv.UDP` (*flags=0, loop=None, on_receive=None*)

Abstraction of UDP sockets for servers and clients.

Raises `uv.UVError` – error while initializing the handle

Parameters

- **flags** (*int*) – udp flags to be used
- **loop** (`uv.Loop`) – event loop the handle should run on
- **on_receive** (*Callable[[`uv.UDP`, `uv.StatusCode`, `uv.Address`, `bytes`, `int`], None]*) – callback called after package has been received

on_receive

Callback called after package has been received.

on_receive (*udp_handle, status, addr, data, flags*)

Parameters

- **udp_handle** (`uv.UDP`) – handle the call originates from
- **status** (`uv.StatusCode`) – status of the handle (indicate any errors)
- **addr** (`uv.Address4` | `uv.Address6` | *tuple*) – address the data originates from
- **data** – data which has been received
- **flags** (*int*) – udp status flags (e.g. partial read)

Readonly `False`

Type Callable[[uv.UDP, uv.StatusCode, uv.Address, bytes, int], None]

open (*fd*)

Open an existing file descriptor as an udp handle.

Raises

- **uv.UVError** – error while opening the handle
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters *fd* (*int*) – file descriptor

bind (*address*, *flags=0*)

Bind the socket to the specified address.

Raises

- **uv.UVError** – error while binding to *address*
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters *address* – address to bind to (*ip*, *port*, *flowinfo=0*, *scope_id=0*)

:param flags bind flags to be used (mask of *uv.UDPFlags*)

send (*buffers*, *address*, *on_send=None*)

Send data over the UDP socket. If the socket has not previously been bound with *bind()* it will be bound to 0.0.0.0 (the “all interfaces” IPv4 address) and a random port number.

Raises

- **uv.UVError** – error while initializing the request
- **uv.ClosedHandleError** – udp handle has already been closed or is closing

Parameters

- **buffers** (*tuple[bytes] | list[bytes] | bytes*) – data which should be send
- **address** (*tuple | uv.Address4 | uv.Address6*) – address tuple (*ip*, *port*, *flowinfo=0*, *scope_id=0*)
- **on_send** (*Callable[[uv.UDPSendRequest, uv.StatusCode], None]:rtype: uv.UDPSendRequest*) – callback called after all data has been sent

try_send (*buffers*, *address*)

Same as *send()*, but won't queue a write request if it cannot be completed immediately.

Raises

- **uv.UVError** – error while sending data
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **buffers** (*tuple[bytes] | list[bytes] | bytes*) – data which should be send

- **address** (*tuple* | *uv.Address4* | *uv.Address6*) – address tuple (*ip*, *port*, *flowinfo=0*, *scope_id=0*)

Returns number of bytes sent

Return type *int*

receive_start (*on_receive=None*)

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

Raises

- **uv.UVError** – error while start receiving datagrams
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters *on_receive* (*Callable*[[*uv.UDP*, *uv.StatusCode*, *uv.Address*, *bytes*, *int*], *None*]) – callback called after package has been received

receive_stop ()

Stop listening for incoming datagrams.

Raises **uv.UVError** – error while stop listening for incoming datagrams

set_membership (*multicast_address*, *membership*, *interface_address=None*)

Set membership for a multicast address

raises uv.UVError: error while setting membership

Raises **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **multicast_address** (*unicode*) – multicast address to set membership for
- **membership** (*uv.UDPMembership*) – membership operation
- **interface_address** (*unicode*) – interface address

set_multicast_loop (*enable*)

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

Raises

- **uv.UVError** – error enabling / disabling multicast loop
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters *enable* (*bool*) – enable / disable multicast loop

set_multicast_ttl (*ttl*)

Set the multicast ttl.

Raises **uv.UVError** – error while setting ttl

:raises uv.ClosedHandleError handle has already been closed or is closing

Parameters *ttl* (*int*) – multicast ttl (between 1 and 255)

set_multicast_interface (*interface*)

Set the multicast interface to send or receive data on.

Raises

- **uv.UVError** – error while setting multicast interface
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **interface** (*unicode*) – multicast interface address

set_broadcast (*enable*)

Set broadcast on or off.

Raises

- **uv.UVError** – error enabling / disabling broadcast
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters **enable** (*bool*) – enable / disable broadcast

family

Address family of UDP handle, may be None.

Readonly True

Return type int | None

sockname

The local IP and port of the UDP handle.

Raises **uv.UVError** – error while receiving sockname

Readonly True

Return type uv.Address4 | uv.Address6

class **uv.UDPFlags**

UDP configuration and status flags enumeration.

IPV6ONLY = None

Disable dual stack support.

Type uv.UDPFlags

REUSEADDR = None

Enable *SO_REUSEADDR* when binding the handle. This sets the *SO_REUSEPORT* socket flag on the BSDs and OSX. On other Unix platforms, it sets the *SO_REUSEADDR* flag. This allows multiple threads or processes to bind to the same address without errors (provided that they all set the flag) but only the last one will receive any traffic, in effect “stealing” the port from the previous listener.

Type uv.UDPFlags

PARTIAL = None

Indicates that the received message has been truncated because the read buffer was too small. The remainder was discarded by the OS.

Type uv.UDPFlags

class **uv.UDPMembership**

Membership types enumeration for multicast addresses.

LEAVE_GROUP = None

Leave multicast group.

Type uv.UDPMembership

JOIN_GROUP = None

Join multicast group.

Type uv.UDPMembership

class `uv.UDPSendRequest` (*udp, buffers, address, on_send=None*)

Request to send a UDP datagram.

Raises

- **uv.UVError** – error while initializing the request
- **uv.ClosedHandleError** – udp handle has already been closed or is closing

Parameters

- **udp** (`uv.UDP`) – udp handle the request should run on
- **buffers** (*list[bytes] | bytes*) – buffers or buffer to send
- **address** (*tuple | uv.Address*) – address of the remote peer (*ip, port, flowinfo=0, scope_id=0*)
- **on_send** (*Callable[[uv.SendRequest, uv.StatusCode], None]*) – callback called after all data has been sent

udp

UDP handle the request belongs to.

Readonly True

Type `uv.UDP`

on_send

Callback which should run after all data has been sent.

Readonly False

Type `Callable[[uv.UDPSendRequest, uv.StatusCode], None]`

1.17 FSEvent – fs event handle

class `uv.FSEvent` (*path=None, flags=0, loop=None, on_event=None*)

FS event handles monitor a given filesystem path for changes including renaming und deletion after they have been started. This handle uses the best backend available for this job on each platform.

Raises **uv.UVError** – error while initializing the handle

Parameters

- **path** (*unicode*) – directory or filename to monitor
- **flags** (*int*) – flags to be used for monitoring
- **loop** (`uv.Loop`) – event loop the handle should run on
- **on_event** (*Callable[[uv.FSEvent, uv.StatusCode, unicode, int], None]*) – callback which should be called on filesystem events after the handle has been started

path

Directory or filename to monitor.

Warning: This property is writable, however you need to restart the handle if you change it during the handle is active.

Readonly False

Type `unicode`

flags

Flags to be used for monitoring.

Warning: This property is writable, however you need to restart the handle if you change it during the handle is active.

Readonly False

Type int

on_event

Callback which should be called on filesystem events after the handle has been started.

on_event (*fs_event*, *status*, *filename*, *events*)

Parameters

- **fs_event** (*uv.FSEvent*) – handle the call originates from
- **status** (*uv.StatusCode*) – may indicate any errors
- **filename** (*unicode*) – if the handle has been started with a directory this will be a relative path to a file contained in that directory which triggered the events
- **events** (*int*) – bitmask of the triggered events

Readonly False

Type Callable[[*uv.FSEvent*, *uv.StatusCode*, *unicode*, *int*], None]

start (*path=None*, *flags=None*, *on_event=None*)

Start watching for filesystem events.

Raises

- **uv.UVError** – error while starting the handle
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **path** (*unicode*) – directory or filename to monitor (overrides the current path if specified)
- **flags** (*int*) – flags to be used for monitoring (overrides the current flags if specified)
- **on_event** (*Callable[[uv.FSEvent, uv.StatusCode, unicode, int], None]*) – callback which should be called on filesystem events (overrides the current callback if specified)

stop ()

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

Raises **uv.UVError** – error while stopping the handle

class *uv.FSEvents*

Events reported by *uv.FSEvent* on filesystem changes.

RENAME = None

File has been renamed or deleted. If the file has been deleted it is necessary (at least on Linux) to restart the corresponding watcher even if the file has been directly recreated.

Type *uv.FSEvents*

CHANGE = None

File has been changed.

Type *uv.FSEvents*

class `uv.FSEventFlags`

Flags to configure the behavior of `uv.FSEvent`.

WATCH_ENTRY = None

By default, if the fs event watcher is given a directory name, it will watch for all events in that directory. This flag overrides this behavior and makes `uv.FSEvent` report only changes to the directory entry itself. This flag does not affect individual files watched.

Note: This flag is currently not implemented yet on any backend.

Type `uv.FSEventFlags`

STAT = None

By default `uv.FSEvent` 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 `uv.FSEvent` fall back to calling `stat()` on a regular interval.

Note: This flag is currently not implemented yet on any backend.

Type `uv.FSEventFlags`

RECURSIVE = None

By default, if the fs event watcher is given a directory name, it will not watch for events in its subdirectories. This flag overrides this behavior and makes `uv.FSEvent` report also changes in subdirectories.

Note: Currently the only supported platforms are OSX and Windows.

Type `uv.FSEventFlags`

1.18 `FSPoll` – fs poll handle

class `uv.FSPoll` (*path=None, interval=5000, loop=None, on_change=None*)

FS poll handles monitor a given filesystem path for changes. Unlike fs event handles, fs poll handles use `stat()` to detect when a file or directory has been changed so they can work on file systems where fs event handles can not.

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

Raises `uv.UVError` – error while initializing the handle

Parameters

- **path** (*unicode*) – directory or filename to monitor
- **interval** (*int*) – interval to be used for monitoring (in milliseconds)
- **loop** (`uv.Loop`) – event loop the handle should run on

- **on_change** (*Callable*[[*uv.FSPoll*, *uv.StatusCode*, *uv.Stat*, *uv.Stat*], *None*]) – callback which should be called on filesystem changes after the handle has been started

path

Directory or filename to monitor.

Warning: This property is writable, however you need to restart the handle if you change it during the handle is active.

Readonly False

Type unicode

interval

Interval to be used for monitoring (in milliseconds).

Warning: This property is writable, however you need to restart the handle if you change it during the handle is active.

Readonly False

Type int

on_change

Callback which should be called on filesystem changes after the handle has been started.

on_change (*fs_poll*, *status*, *previous_stat*, *current_stat*)

Parameters

- **fs_event** (*uv.FSEvent*) – handle the call originates from
- **status** (*uv.StatusCode*) – may indicate any errors
- **previous_stat** (*uv.Stat*) – previous filesystem path’s stat
- **current_stat** (*uv.Stat*) – current filesystem path’s stat

Readonly False

Type Callable[[*uv.FSPoll*, *uv.StatusCode*, *uv.Stat*, *uv.Stat*], *None*]

start (*path=None*, *interval=None*, *on_change=None*)

Start monitoring for filesystem changes. The change callback is invoked with status code < 0 if the given 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).

Raises

- **uv.UVError** – error while starting the handle
- **uv.ClosedHandleError** – handle has already been closed or is closing

Parameters

- **path** (*unicode*) – directory or filename to monitor (overrides the current path if specified)
- **interval** (*int*) – interval to be used for monitoring (in milliseconds and overrides the current interval if specified)
- **on_change** (*Callable*[[*uv.FSPoll*, *uv.StatusCode*, *uv.Stat*, *uv.Stat*], *None*]) – callback which should be called on filesystem changes

stop ()

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

Raises `uv.UVError` – error while stopping the handle

1.19 Request – request base classes

class `uv.Request`

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. This is the abstract base class of all internal libuv and pure Python requests.

cancel ()

Cancel a pending request. Fails if the request is executing or has finished executing.

class `uv.UVRequest` (*loop, arguments, uv_handle=None, request_init=None*)

The base class of all libuv based requests.

Raises `uv.LoopClosedError` – loop has already been closed

Parameters

- **loop** (`Loop`) – loop where the request should run on
- **arguments** (`tuple`) – arguments passed to the libuv request initializer
- **uv_handle** (`ffi.CData`) – libuv handle the requests belongs to
- **request_init** (`callable`) – libuv function for request initialization

loop

Loop where the request is running on.

Readonly True

Type `Loop`

finished

Request has been finished.

Readonly True

Type `bool`

type

Type of the request. Returns a subclass of `uv.UVRequest`.

Type `type`

cancel ()

Raises `uv.UVError` – error while canceling request

set_pending ()

Warning: This method is only for internal purposes and is not part of the official API.

clear_pending ()

Warning: This method is only for internal purposes and is not part of the official API.

1.20 DNS utilities

`uv.getaddrinfo` (*host, port, family=0, socktype=0, protocol=0, flags=0, callback=None, loop=None*)

Get address information for specified host and port (service).

See `uv.dns.GetAddrInfo` for parameter descriptions.

Return type `uv.GetAddrInfo` | `list[uv.AddrInfo]`

`uv.getnameinfo` (*ip, port, flags=0, callback=None, loop=None*)

Get name information for specified IP and port.

See `uv.dns.GetNameInfo` for parameter descriptions.

Return type `uv.GetNameInfo` | `uv.NameInfo`

1.20.1 Requests

class `uv.dns.GetAddrInfo` (*host, port, family=0, socktype=0, protocol=0, flags=0, callback=None, loop=None*)

Request to get address information for specified host and port (service). If no callback is provided the request is executed synchronously.

Raises `uv.UVError` – error while initializing the handle

Parameters

- **host** (*unicode*) – host to get address information for
- **port** (*int*) – port (service) to get address information for
- **family** (*uv.AddressFamilies* | *int*) – address family hint
- **socktype** (*uv.SocketTypes* | *int*) – socket type hint
- **protocol** (*uv.SocketProtocols* | *int*) – protocol type hint
- **flags** (*int*) – flags to configure the behavior of `getaddrinfo()`
- **callback** – callback which should be called after address information has been fetched or on error
- **loop** (*uv.Loop*) – event loop the request should run on

callback

Callback which should be called after address information has been fetched or on error.

callback (*request, code, addrinfo*)

Parameters

- **request** (*uv.GetAddrInfo*) – request the call originates from
- **code** (*uv.StatusCodes*) – status of the request
- **addrinfo** (*list[uv.AddrInfo]*) – list of fetched address information objects

Readonly `False`

Type

`((uv.GetAddrInfo, uv.StatusCodes, list[uv.AddrInfo]) -> None)` |

`((Any, uv.GetAddrInfo, uv.StatusCodes, list[uv.AddrInfo]) -> None)`

host

Host to get address information for.

Readonly True**Type** unicode**port**

Port (service) to get address information for.

Readonly True**Type** int**hints**

Address information hints.

Readonly True**Type** uv.AddrInfo**flags**Flags to configure the behavior of *getaddrinfo()*.**Readonly** True**Type** int**addrinfo**

Resulting list with address information. Is populated with address information objects after the request has been successfully completed.

Readonly True**Type** list[uv.AddrInfo]**class** uv.dns.**GetNameInfo** (*ip, port, flags=0, callback=None, loop=None*)

Request to get name information for specified ip and port. If no callback is provided the request is executed synchronously.

Parameters

- **ip** (*unicode*) – IP to get name information for
- **port** (*int*) – port to get name information for
- **flags** (*int*) – flags to configure the behavior of *getnameinfo()*
- **callback** (*Callable[[uv.GetNameInfo, uv.StatusCodes, unicode, unicode], None]None*) – callback which should be called after name information has been fetched or on error
- **loop** (*uv.Loop*) – event loop the request should run on

callback

Callback which should be called after name information has been fetched or on error.

callback (*request, code, hostname, service*)**Parameters**

- **request** (*uv.GetNameInfo*) – request the call originates from
- **code** (*uv.StatusCodes*) – status of the request
- **hostname** (*unicode*) – hostname corresponding to the given IP
- **service** (*unicode*) – service name corresponding to the given port

Readonly False**Type** Callable[[uv.GetNameInfo, uv.StatusCodes, unicode, unicode], None]

ip
Ip to get name information for.
Readonly True
Type unicode

port
Port to get name information for.
Readonly True
Type int

flags
Flags to configure the behavior of *getnameinfo()*.
Readonly True
Type int

hostname
Hostname corresponding to the given IP.
Readonly True
Type unicode

service
Service name corresponding to the given port.
Readonly True
Returns unicode

1.20.2 Data Structures

class `uv.Address`
Internet Protocol Address

host
Address Host
Return type unicode

port
Address Port
Return type int

class `uv.Address4`
Internet protocol version 4 address.
family = `<AddressFamilies.INET4: 2>`
Address Family

class `uv.Address6`
Internet protocol version 6 address.
family = `<AddressFamilies.INET6: 10>`
Address Family

flowinfo
Address Flow Information

Return type int**scope_id**

Address Scope Id

Return type int**class** uv.**AddrInfo**

Address Information

family

Address Family

Return type uv.AddressFamilies | int**socktype**

Socket Type

Return type uv.SocketTypes | int**protocol**

Socket Protocol

Return type uv.SocketProtocols | int**canonicalname**

Canonical Name

Return type unicode | None**address**

Address

Return type uv.Address | tuple | None**class** uv.**NameInfo**

Name Information

hostname

Hostname

Return type unicode**service**

Service-Name

Return type unicode

1.20.3 Enumerations

class uv.**AddressFamilies**

Address Families

UNKNOWN = None

Unknown or unspecified family.

Type uv.AddressFamilies**INET4 = None**

Internet protocol version 4 family.

Type uv.AddressFamilies**INET6 = None**

Internet protocol version 6 family.

Type uv.AddressFamilies

UNIX = None

Unix domain sockets family.

Type uv.AddressFamilies

class uv.**SocketTypes**

Socket Types

STREAM = None

Stream socket type.

Type uv.SocketTypes

DRGRAM = None

Datagram socket type.

Type uv.SocketTypes

RAW = None

Raw socket type.

Type uv.SocketTypes

RDM = None

Pragmatic general multicast socket type.

Type uv.SocketTypes

SEQPACKET = None

Sequence packet socket type.

Type uv.SocketTypes

class uv.**SocketProtocols**

Socket Protocols

TCP = None

Transmission control protocol.

Type uv.SocketProtocols

UDP = None

User datagram protocol.

Type uv.SocketProtocols

RAW = None

Raw socket protocol.

Type uv.SocketProtocols

ICMP4 = None

Internet version 4 control message protocol.

Type uv.SocketProtocols

ICMP6 = None

Internet version 6 control message protocol.

Type uv.SocketProtocols

Indices and tables

- `genindex`
- `modindex`
- `search`

u

`uv.error`, 1

A

accept() (uv.UVStream method), 34
active (uv.Handle attribute), 16
active (uv.UVHandle attribute), 18
Address (class in uv), 50
address (uv.AddrInfo attribute), 51
Address4 (class in uv), 50
Address6 (class in uv), 50
AddressDataError, 9
AddressError, 9
AddressFamilies (class in uv), 51
AddressFamilyError, 9
AddressFlagsError, 9
AddressHintsError, 9
AddressInUseError, 9
AddressNameError, 9
AddressProtocolError, 10
AddressServiceError, 10
AddressSocketTypeError, 10
AddressUnavailableError, 9
AddrInfo (class in uv), 51
addrinfo (uv.dns.GetAddrInfo attribute), 49
again() (uv.Timer method), 30
alive (uv.Loop attribute), 12
allocate() (uv.loop.Allocator method), 15
Allocator (class in uv.loop), 15
allocator (uv.UVHandle attribute), 17
ArgumentError, 7
Async (class in uv), 19
Async.on_wakeup() (in module uv), 20

B

BadFileDescriptorError, 8
bind() (uv.Pipe method), 22
bind() (uv.TCP method), 36
bind() (uv.UDP method), 40
BrokenPipeError, 10
BufferSpaceError, 8

C

call_later() (uv.Loop method), 14
callback (uv.dns.GetAddrInfo attribute), 48
callback (uv.dns.GetNameInfo attribute), 49
cancel() (uv.Request method), 47
cancel() (uv.UVRequest method), 47
CanceledError, 7
canonname (uv.AddrInfo attribute), 51
CHANGE (uv.FSEvents attribute), 44
CharsetError, 8
Check (class in uv), 20
Check.on_check() (in module uv), 20
clear_pending() (uv.UVHandle method), 19
clear_pending() (uv.UVRequest method), 47
close() (uv.Handle method), 17
close() (uv.Loop method), 13
close() (uv.UVHandle method), 19
close_all_handles() (uv.Loop method), 13
closed (uv.Handle attribute), 16
closed (uv.Loop attribute), 12
closed (uv.UVHandle attribute), 18
ClosedHandleError, 9
ClosedLoopError, 9
ClosedStructureError, 9
closing (uv.Handle attribute), 16
closing (uv.UVHandle attribute), 17
code (uv.error.UVError attribute), 7
connect() (uv.Pipe method), 22
connect() (uv.TCP method), 37
ConnectionAbortedError, 10
ConnectionError, 10
ConnectionInProgressError, 10
ConnectionRefusedError, 10
ConnectionResetError, 10
ConnectRequest (class in uv), 34
console_size (uv.TTY attribute), 38
ConsoleSize (class in uv), 39
CreatePipe (class in uv), 27
CrossDeviceError, 9

D

data (uv.UVHandle attribute), 17
 DEFAULT (uv.RunModes attribute), 14
 DefaultAllocator (class in uv.loop), 15
 dereference() (uv.Handle method), 16
 dereference() (uv.UVHandle method), 19
 DestinationAddressError, 10
 DETACHED (uv.ProcessFlags attribute), 27
 DeviceNotFoundError, 10
 DRGAM (uv.SocketTypes attribute), 52

E

E2BIG (uv.error.StatusCodes attribute), 1
 EACCES (uv.error.StatusCodes attribute), 1
 EADDRINUSE (uv.error.StatusCodes attribute), 1
 EADDRNOTAVAIL (uv.error.StatusCodes attribute), 1
 EAFNOSUPPORT (uv.error.StatusCodes attribute), 1
 EAGAIN (uv.error.StatusCodes attribute), 1
 EAI_ADDRFAMILY (uv.error.StatusCodes attribute), 1
 EAI_AGAIN (uv.error.StatusCodes attribute), 1
 EAI_BADFLAGS (uv.error.StatusCodes attribute), 2
 EAI_BADHINTS (uv.error.StatusCodes attribute), 2
 EAI_CANCELED (uv.error.StatusCodes attribute), 2
 EAI_FAIL (uv.error.StatusCodes attribute), 2
 EAI_FAMILY (uv.error.StatusCodes attribute), 2
 EAI_MEMORY (uv.error.StatusCodes attribute), 2
 EAI_NODATA (uv.error.StatusCodes attribute), 2
 EAI_NONAME (uv.error.StatusCodes attribute), 2
 EAI_OVERFLOW (uv.error.StatusCodes attribute), 2
 EAI_PROTOCOL (uv.error.StatusCodes attribute), 2
 EAI_SERVICE (uv.error.StatusCodes attribute), 2
 EAI_SOCKTYPE (uv.error.StatusCodes attribute), 2
 EALREADY (uv.error.StatusCodes attribute), 2
 EBADF (uv.error.StatusCodes attribute), 3
 EBUSY (uv.error.StatusCodes attribute), 3
 ECANCELED (uv.error.StatusCodes attribute), 3
 ECHARSET (uv.error.StatusCodes attribute), 3
 ECONNABORTED (uv.error.StatusCodes attribute), 3
 ECONNREFUSED (uv.error.StatusCodes attribute), 3
 ECONNRESET (uv.error.StatusCodes attribute), 3
 EDESTADDRREQ (uv.error.StatusCodes attribute), 3
 EEXIST (uv.error.StatusCodes attribute), 3
 EFAULT (uv.error.StatusCodes attribute), 3
 EFBIG (uv.error.StatusCodes attribute), 3
 EHOSTDOWN (uv.error.StatusCodes attribute), 7
 EHOSTUNREACH (uv.error.StatusCodes attribute), 3
 EINTR (uv.error.StatusCodes attribute), 3
 EINVAL (uv.error.StatusCodes attribute), 3
 EIO (uv.error.StatusCodes attribute), 4
 EISCONN (uv.error.StatusCodes attribute), 4
 EISDIR (uv.error.StatusCodes attribute), 4
 ELOOP (uv.error.StatusCodes attribute), 4
 EMFILE (uv.error.StatusCodes attribute), 4
 EMLINK (uv.error.StatusCodes attribute), 6

EMSGSIZE (uv.error.StatusCodes attribute), 4
 ENAMETOOLONG (uv.error.StatusCodes attribute), 4
 ENETDOWN (uv.error.StatusCodes attribute), 4
 ENETUNREACH (uv.error.StatusCodes attribute), 4
 ENFILE (uv.error.StatusCodes attribute), 4
 ENOBUFS (uv.error.StatusCodes attribute), 4
 ENODEV (uv.error.StatusCodes attribute), 4
 ENOENT (uv.error.StatusCodes attribute), 4
 ENOMEM (uv.error.StatusCodes attribute), 5
 ENONET (uv.error.StatusCodes attribute), 5
 ENOPROTOOPT (uv.error.StatusCodes attribute), 5
 ENOSPC (uv.error.StatusCodes attribute), 5
 ENOSYS (uv.error.StatusCodes attribute), 5
 ENOTCONN (uv.error.StatusCodes attribute), 5
 ENOTDIR (uv.error.StatusCodes attribute), 5
 ENOTEMPTY (uv.error.StatusCodes attribute), 5
 ENOTSOCK (uv.error.StatusCodes attribute), 5
 ENOTSUP (uv.error.StatusCodes attribute), 5
 ENXIO (uv.error.StatusCodes attribute), 6
 EOF (uv.error.StatusCodes attribute), 6
 EOFError, 9
 EPERM (uv.error.StatusCodes attribute), 5
 EPIPE (uv.error.StatusCodes attribute), 5
 EPROTO (uv.error.StatusCodes attribute), 5
 EPROTONOSUPPORT (uv.error.StatusCodes attribute),
 5
 EPROTOTYPE (uv.error.StatusCodes attribute), 6
 ERANGE (uv.error.StatusCodes attribute), 6
 EROFS (uv.error.StatusCodes attribute), 6
 ESHUTDOWN (uv.error.StatusCodes attribute), 6
 ESPIPE (uv.error.StatusCodes attribute), 6
 ESRCH (uv.error.StatusCodes attribute), 6
 ETIMEDOUT (uv.error.StatusCodes attribute), 6
 ETXTBSY (uv.error.StatusCodes attribute), 6
 exc_traceback (uv.Loop attribute), 12
 exc_type (uv.Loop attribute), 12
 exc_value (uv.Loop attribute), 12
 excepthook (uv.Loop attribute), 11
 exception (uv.error.StatusCodes attribute), 7
 EXDEV (uv.error.StatusCodes attribute), 6

F

family (uv.Address4 attribute), 50
 family (uv.Address6 attribute), 50
 family (uv.AddrInfo attribute), 51
 family (uv.UDP attribute), 42
 family (uv.UVStream attribute), 33
 fd (uv.Poll attribute), 24
 FileExistsError, 8
 fileno() (uv.Loop method), 12
 fileno() (uv.Poll method), 24
 fileno() (uv.UVHandle method), 18
 FileNotFoundError, 10
 FileTableOverflowError, 11

FileTooLargeError, 8
 finalize() (uv.loop.Allocator method), 15
 finished (uv.UVRequest attribute), 47
 flags (uv.dns.GetAddrInfo attribute), 49
 flags (uv.dns.GetNameInfo attribute), 50
 flags (uv.FSEvent attribute), 44
 flowinfo (uv.Address6 attribute), 50
 FSEvent (class in uv), 43
 FSEvent.on_event() (in module uv), 44
 FSEventFlags (class in uv), 44
 FSEvents (class in uv), 44
 FSPoll (class in uv), 45
 FSPoll.on_change() (in module uv), 46

G

get() (uv.error.StatusCodes class method), 7
 get_current() (uv.Loop class method), 11
 get_default() (uv.Loop class method), 11
 get_timeout() (uv.Loop method), 13
 GetAddrInfo (class in uv.dns), 48
 getaddrinfo() (in module uv), 48
 GetAddrInfo.callback() (in module uv), 48
 GetNameInfo (class in uv.dns), 49
 getnameinfo() (in module uv), 48
 GetNameInfo.callback() (in module uv), 49

H

Handle (class in uv), 16
 handle_exception() (uv.Loop method), 14
 handles (uv.Loop attribute), 12
 height (uv.ConsoleSize attribute), 39
 hints (uv.dns.GetAddrInfo attribute), 49
 host (uv.Address attribute), 50
 host (uv.dns.GetAddrInfo attribute), 48
 hostname (uv.dns.GetNameInfo attribute), 50
 hostname (uv.NameInfo attribute), 51
 HostUnreachableError, 8

I

ICMP4 (uv.SocketProtocols attribute), 52
 ICMP6 (uv.SocketProtocols attribute), 52
 Idle (class in uv), 21
 Idle.on_idle() (in module uv), 21
 INET4 (uv.AddressFamilies attribute), 51
 INET6 (uv.AddressFamilies attribute), 51
 InterruptedError, 8
 interval (uv.FSPoll attribute), 46
 IO (uv.TTYMode attribute), 39
 IOError, 8
 ip (uv.dns.GetNameInfo attribute), 49
 ipc (uv.UVStream attribute), 33
 IPV6ONLY (uv.TCPFlags attribute), 38
 IPV6ONLY (uv.UDPFlags attribute), 42
 IsADirectoryError, 8

IsConnectedError, 8

J

JOIN_GROUP (uv.UDPMembership attribute), 42

K

kill() (uv.Process method), 27

L

LEAVE_GROUP (uv.UDPMembership attribute), 42

listen() (uv.UVStream method), 33

Loop (class in uv), 11

loop (uv.Handle attribute), 16

loop (uv.UVRequest attribute), 47

Loop.excepthook() (in module uv), 11

M

make_current() (uv.Loop method), 13

MessageTooLongError, 8

MemoryError, 10

message (uv.error.StatusCodes attribute), 7

message (uv.error.UVError attribute), 7

N

name (uv.error.UVError attribute), 7

NameInfo (class in uv), 51

NameTooLongError, 8

NetworkDownError, 10

NetworkError, 10

NetworkUnreachableError, 10

NoNetworkError, 10

NORMAL (uv.TTYMode attribute), 38

NoSpaceError, 8

NotADirectoryError, 8

NotConnectedError, 8

NotEmptyError, 8

NotFoundError, 10

NotImplementedError, 8

NotSocketError, 9

NotSupportedError, 9

now (uv.Loop attribute), 12

NOWAIT (uv.RunModes attribute), 15

O

on_change (uv.FSPoll attribute), 46

on_check (uv.Check attribute), 20

on_closed (uv.UVHandle attribute), 17

on_connect (uv.ConnectRequest attribute), 34

on_connection (uv.UVStream attribute), 32

on_event (uv.FSEvent attribute), 44

on_event (uv.Poll attribute), 24

on_exit (uv.Process attribute), 27

on_idle (uv.Idle attribute), 21

[on_prepare \(uv.Prepare attribute\)](#), 25
[on_read \(uv.UVStream attribute\)](#), 32
[on_receive \(uv.UDP attribute\)](#), 39
[on_send \(uv.UDPSendRequest attribute\)](#), 43
[on_shutdown \(uv.ShutdownRequest attribute\)](#), 36
[on_signal \(uv.Signal attribute\)](#), 28
[on_timeout \(uv.Timer attribute\)](#), 30
[on_wakeup \(uv.Async attribute\)](#), 20
[on_wakeup\(\) \(uv.Loop method\)](#), 14
[on_write \(uv.WriteRequest attribute\)](#), 35
[ONCE \(uv.RunModes attribute\)](#), 15
[open\(\) \(uv.Pipe method\)](#), 22
[open\(\) \(uv.TCP method\)](#), 36
[open\(\) \(uv.UDP method\)](#), 40

P

[PARTIAL \(uv.UDPFlags attribute\)](#), 42
[path \(uv.FSEvent attribute\)](#), 43
[path \(uv.FSPoll attribute\)](#), 46
[peername \(uv.Pipe attribute\)](#), 23
[peername \(uv.TCP attribute\)](#), 37
[pending_accept\(\) \(uv.Pipe method\)](#), 22
[pending_count \(uv.Pipe attribute\)](#), 22
[pending_instances\(\) \(uv.Pipe method\)](#), 23
[pending_type \(uv.Pipe attribute\)](#), 22
[PermanentError](#), 7
[PermissionError](#), 7
[pid \(uv.Process attribute\)](#), 27
[Pipe \(class in uv\)](#), 21
[PIPE \(in module uv\)](#), 27
[PipeConnectRequest \(class in uv\)](#), 23
[Poll \(class in uv\)](#), 23
[Poll.on_event\(\) \(in module uv\)](#), 24
[PollEvent \(class in uv\)](#), 25
[port \(uv.Address attribute\)](#), 50
[port \(uv.dns.GetAddrInfo attribute\)](#), 49
[port \(uv.dns.GetNameInfo attribute\)](#), 50
[Prepare \(class in uv\)](#), 25
[Prepare.on_prepare\(\) \(in module uv\)](#), 25
[Process \(class in uv\)](#), 26
[Process.on_exit\(\) \(in module uv\)](#), 27
[ProcessFlags \(class in uv\)](#), 27
[ProcessLookupError](#), 8
[protocol \(uv.AddrInfo attribute\)](#), 51
[ProtocolError](#), 9
[ProtocolNoOptionError](#), 9
[ProtocolNotSupportedError](#), 9
[ProtocolTypeError](#), 9

R

[RAW \(uv.SocketProtocols attribute\)](#), 52
[RAW \(uv.SocketTypes attribute\)](#), 52
[RAW \(uv.TTYMode attribute\)](#), 39
[RDM \(uv.SocketTypes attribute\)](#), 52

[read_start\(\) \(uv.Stream method\)](#), 31
[read_start\(\) \(uv.UVStream method\)](#), 33
[read_stop\(\) \(uv.Stream method\)](#), 31
[read_stop\(\) \(uv.UVStream method\)](#), 33
[READABLE \(uv.PollEvent attribute\)](#), 25
[readable \(uv.Stream attribute\)](#), 31
[readable \(uv.UVStream attribute\)](#), 33
[receive_buffer_size \(uv.UVHandle attribute\)](#), 18
[receive_start\(\) \(uv.UDP method\)](#), 41
[receive_stop\(\) \(uv.UDP method\)](#), 41
[RECURSIVE \(uv.FSEventFlags attribute\)](#), 45
[reference\(\) \(uv.Handle method\)](#), 16
[reference\(\) \(uv.UVHandle method\)](#), 19
[referenced \(uv.Handle attribute\)](#), 16
[referenced \(uv.UVHandle attribute\)](#), 18
[RENAME \(uv.FSEvents attribute\)](#), 44
[repeat \(uv.Timer attribute\)](#), 30
[Request \(class in uv\)](#), 47
[reset_exception\(\) \(uv.Loop method\)](#), 14
[ResourceBusyError](#), 8
[ResultTooLargeError](#), 8
[REUSEADDR \(uv.UDPFlags attribute\)](#), 42
[run\(\) \(uv.Loop method\)](#), 13
[RunModes \(class in uv\)](#), 14

S

[scope_id \(uv.Address6 attribute\)](#), 51
[SeekError](#), 8
[send\(\) \(uv.Async method\)](#), 20
[send\(\) \(uv.UDP method\)](#), 40
[send_buffer_size \(uv.UVHandle attribute\)](#), 18
[send_stream \(uv.WriteRequest attribute\)](#), 35
[SEQPACKET \(uv.SocketTypes attribute\)](#), 52
[service \(uv.dns.GetNameInfo attribute\)](#), 50
[service \(uv.NameInfo attribute\)](#), 51
[set_broadcast\(\) \(uv.UDP method\)](#), 42
[set_keepalive\(\) \(uv.TCP method\)](#), 37
[set_membership\(\) \(uv.UDP method\)](#), 41
[set_mode\(\) \(uv.TTY method\)](#), 38
[set_multicast_interface\(\) \(uv.UDP method\)](#), 41
[set_multicast_loop\(\) \(uv.UDP method\)](#), 41
[set_multicast_ttl\(\) \(uv.UDP method\)](#), 41
[set_nodelay\(\) \(uv.TCP method\)](#), 37
[set_pending\(\) \(uv.UVHandle method\)](#), 19
[set_pending\(\) \(uv.UVRequest method\)](#), 47
[set_simultaneous_accepts\(\) \(uv.TCP method\)](#), 38
[shutdown\(\) \(uv.Stream method\)](#), 32
[shutdown\(\) \(uv.UVStream method\)](#), 33
[ShutdownRequest \(class in uv\)](#), 35
[ShutdownRequest.on_shutdown\(\) \(in module uv\)](#), 36
[SIGBREAK \(uv.Signals attribute\)](#), 29
[SIGHUP \(uv.Signals attribute\)](#), 29
[SIGINT \(uv.Signals attribute\)](#), 29
[Signal \(class in uv\)](#), 28

Signals (class in uv), 29
 signum (uv.Signal attribute), 28
 SIGWINCH (uv.Signals attribute), 29
 SocketProtocols (class in uv), 52
 SocketTypes (class in uv), 52
 sockname (uv.Pipe attribute), 23
 sockname (uv.TCP attribute), 37
 sockname (uv.UDP attribute), 42
 socktype (uv.AddrInfo attribute), 51
 start() (uv.Check method), 20
 start() (uv.FSEvent method), 44
 start() (uv.FSPoll method), 46
 start() (uv.Idle method), 21
 start() (uv.Poll method), 24
 start() (uv.Prepare method), 25
 start() (uv.Signal method), 29
 start() (uv.Timer method), 30
 STAT (uv.FSEventFlags attribute), 45
 StatusCodes (class in uv.error), 1
 stderr (uv.Process attribute), 27
 stdin (uv.Process attribute), 26
 stdio (uv.Process attribute), 27
 stdout (uv.Process attribute), 26
 stop() (uv.Check method), 21
 stop() (uv.FSEvent method), 44
 stop() (uv.FSPoll method), 46
 stop() (uv.Idle method), 21
 stop() (uv.Loop method), 13
 stop() (uv.Poll method), 25
 stop() (uv.Prepare method), 26
 stop() (uv.Signal method), 29
 stop() (uv.Timer method), 31
 Stream (class in uv), 31
 stream (uv.ConnectRequest attribute), 34
 stream (uv.ShutdownRequest attribute), 35
 STREAM (uv.SocketTypes attribute), 52
 stream (uv.WriteRequest attribute), 35
 structure_clear_pending() (uv.Loop method), 14
 structure_is_pending() (uv.Loop method), 14
 structure_set_pending() (uv.Loop method), 14
 SUCCESS (uv.error.StatusCodes attribute), 1
 SystemFailureError, 10

T

TCP (class in uv), 36
 TCP (uv.SocketProtocols attribute), 52
 TCPFlags (class in uv), 38
 TemporaryUnavailableError, 7
 TimeoutError, 9
 Timer (class in uv), 29
 Timer.on_timeout() (in module uv), 30
 TooManyLinksError, 10
 TooManyOpenFilesError, 11
 TooManySymbolicLinksError, 10

try_send() (uv.UDP method), 40
 try_write() (uv.UVStream method), 33
 TTY (class in uv), 38
 TTYMode (class in uv), 38
 type (uv.UVRequest attribute), 47

U

UDP (class in uv), 39
 UDP (uv.SocketProtocols attribute), 52
 udp (uv.UDPSendRequest attribute), 43
 UDP.on_receive() (in module uv), 39
 UDPFlags (class in uv), 42
 UDPMembership (class in uv), 42
 UDPSendRequest (class in uv), 42
 UNIX (uv.AddressFamilies attribute), 52
 UNKNOWN (uv.AddressFamilies attribute), 51
 UNKNOWN (uv.error.StatusCodes attribute), 6
 UnsupportedOperation, 9
 update_time() (uv.Loop method), 13
 uv.error (module), 1
 UVError, 7
 UVHandle (class in uv), 17
 UVHandle.on_closed() (in module uv), 17
 UVRequest (class in uv), 47
 UVStream (class in uv), 32
 UVStream.on_connection() (in module uv), 32
 UVStream.on_read() (in module uv), 32

W

WATCH_ENTRY (uv.FSEventFlags attribute), 45
 width (uv.ConsoleSize attribute), 39
 WINDOWS_HIDE (uv.ProcessFlags attribute), 28
 WINDOWS_VERBATIM (uv.ProcessFlags attribute), 28
 WRITABLE (uv.PollEvent attribute), 25
 writable (uv.UVStream attribute), 33
 write() (uv.Stream method), 31
 write() (uv.UVStream method), 33
 writeable (uv.Stream attribute), 31
 WriteRequest (class in uv), 35