Python libuv CFFI Bindings

Release 0.1.0.dev0

February 13, 2016

Contents

CHAPTER 1

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 occoured.

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

ENOPROTOOPT = 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]

name
Human readable error name.

Readonly True

Return type unicode

message
Human readable error message.

Readonly True

Return type unicode

classmethod get(*code*)
Look up the given status code und 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.**IOError**(*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.**MassageTooLongError**(*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.

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.

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

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 class

class uv.**Handle**(*loop*, *arguments=()*)
Handles represent long-lived objects capable of performing certain operations while active. This is the base class of all handles except the file and SSL handle, which are pure Python.

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.

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

loop
Loop the handle is running on.

Readonly True

Type uv.Loop

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
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

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

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

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()

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 not 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)

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 given callback in the event loop's thread without creating an *uv.Async* handle use *uv.Loop.call_later()*.

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 form 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.

on_check

Callback which should run right after polling for IO if 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.

on_idle
Callback which should run right before the prepare handles.

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.

open(*fd*)
Open an existing file descriptor as a pipe.

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.Stream*.

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.Stream*

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.

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.

---

fd
File descriptor the handle polls on.

Readonly True

Type int

on_event
Callback which should be called on IO events.

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.

on_prepare
Callback which should run right before polling for IO.

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.

stdin = None
Standard input of the child process.

Readonly True

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

stdout = None
Standard output of the child process.

Readonly True

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

stderr = None
Standard error of the child process.

Readonly True

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

stdio = None
Other standard file descriptors of the child process.

Readonly True

Type list[int | uv.Stream | 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.

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. 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.

---

on_signal

Callback which should be called on signal delivery.

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.

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**(*loop*, *ipc*, *arguments*, *on_read*, *on_connection*)

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

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

---

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

Readonly False

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

ipc
Stream does support inter process communication or not.

Readonly True

Type bool

readable
Stream is readable or not.

Readonly True

Type bool

writable
Stream is writable or not.

Readonly True

Type bool

family
Address family of stream, may be None.

Return type int | None

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.ShutdownRequest*

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.Stream, 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*)
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.WriteRequest, uv.StatusCodes], None]:returns: issued write request*) – callback which should run after all data has been written

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.Stream*

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.

stream
Stream to establish a connection on.

Readonly True

Type uv.Stream

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.

stream
Stream to write data to.

Readonly True

Type uv.Stream

send_stream
Stream handle which should be send.

Readonly True

Type uv.Stream | 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.

stream
Stream to shutdown.

Readonly True

Type uv.Stream

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.

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.

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

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.

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.

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.

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.

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

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

u

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W