Python libuv CFFI Bindings

Release 0.0.4.dev0

October 10, 2016

Contents

CHAPTER 1

Contents:

1.1 Errors – exceptions and error handling

class `uv.`**`StatusCode`**

`SUCCESS` **= None**
Success.

class `uv.`**`UVError`**(*code*, *message=None*)

class `uv.`**`HandleClosedError`**

class `uv.`**`LoopClosedError`**

1.2 Handle – handle base class

class `uv.`**`Handle`**(*uv_handle*, *loop=None*)
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.

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

Parameters

- **`loop`** (*`Loop`*) – loop where the handle should run on
- **`uv_handle`** (*`ffi.CData`*) – allocated c struct for this handle

`loop`
Loop where the handle is running on.

Readonly True

Type Loop

`on_closed`
Callback which should be called after the handle has been closed.

Readonly False

Type (Handle) -> None

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

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.HandleClosedError`.

Readonly True

Type bool

active

Handle is active or not. What "active" means depends on the handle:

- `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 *StatusCode.EINVAL*.

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.HandleClosedError`** – 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 *StatusCode.EINVAL*.

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.HandleClosedError` – handle has already been closed or is closing

Readonly False

Type int

fileno()

Gets 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 *StatusCode.EINVAL*.

If a handle does not have an attached file descriptor yet this method will raise `uv.UVError` with *StatusCode.EBADF*.

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.HandleClosedError` – handle has already been closed or is closing

Returns platform dependent file descriptor equivalent

Return type int

reference()

References 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 already referenced calling this method again will have not effect.

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

dereference()

Dereferences 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.HandleClosedError` – handle has already been closed or is closing

close(*on_closed=None*)

Closes the handle and frees all resources afterwards. Please make sure to call this method on any handle you do not need anymore. Handles do not close automatically and are also not garbage collected unless you have closed them exlicitly (explicit is better than implicit). This method is idempotent, that is, if the handle is already closed or is closing calling this method will have no effect.

In-progress requests, like `uv.ConnectRequest` or `uv.WriteRequest`, are cancelled and have their callbacks called asynchronously with `StatusCode.ECANCELED`

After this method has been called on a handle no other operations can be performed on it, they will raise `uv.HandleClosedError`.

Parameters **on_closed** (`(Handle) -> None`) – callback called after the handle has been closed

destroy()

> **Warning:** This method is used internally to free all allocated C resources and make sure there are no references from Python anymore to those objects after the handle has been closed. You should never call it directly!

1.3 `Async` – async handle

class uv.**Async**(*loop=None*, *callback=None*)

Async handles will wake-up the event loop from an other thread and run the given callback within the event loop's thread. They are the only thread-safe handles.

Raises ***uv.UVError*** – error during the initialization of the handle

Parameters

- **loop** (*uv.Loop*) – event loop which should be used for the handle
- **callback** (*(uv.Async) -> None*) – callback which should be called from within the event loop

callback

Callback which should be called from within the event loop.

callback(*Async-Handle*)

Readonly False

Type (uv.Async) -> None

send(*callback=None*)

Wake-up the event loop and execute 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 a execution of the callback.

Raises

- ***uv.UVError*** – error while trying to wake-up event loop
- ***uv.HandleClosedError*** – handle has already been closed or is closing

Parameters **callback** (*(uv.Async) -> None*) – callback which should be called from within the event loop

1.4 `Check` – check handle

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

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

Raises ***uv.UVError*** – error during the initialization of the handle

Parameters

- **loop** (*Loop*) – event loop which should be used for the handle
- **callback** (*(uv.Check) -> None*) – callback which should be called right after polling for IO

callback
Callback which should be called after polling for IO.

callback (*Check-Handle*)

Readonly False

Type (uv.Check) -> None

start (*callback=None*)
Starts the handle.

Raises

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

Parameters callback (*(uv.Check) -> None*) – callback which should be called after polling for IO

stop ()
Stops the handle, the callback will no longer be called.

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

1.5 `Idle` – idle handle

class uv.**Idle** (*loop=None*, *callback=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 during the initialization of the handle

Parameters

- **loop** (*uv.Loop*) – event loop which should be used for the handle
- **callback** (*(uv.Idle) -> None*) – callback which should be called before prepare handles

callback
Callback which should be called before prepare handles.

callback (*Idle-Handle*)

Readonly False

Type (uv.Idle) -> None

start (*callback=None*)
Starts the handle.

Raises

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

Parameters **`callback`** (*`(uv.Idle) -> None`*) – callback which should be called before prepare handles

`stop`()

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

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

1.6 `Poll` – poll handle

class `uv.`**`Poll`**(*fd*, *loop=None*, *callback=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 an more scalable implementations, that 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 during the initialization of the handle

Parameters

- **`fd`** (*`int`*) – file descriptor which should be polled (is set to non-blocking mode)
- **`loop`** (*`Loop`*) – event loop which should be used for the handle
- **`callback`** (*`(uv.Poll, uv.StatusCode, int) -> None`*) – callback which should be called on IO events

`fd`

File descriptor the handle polls on.

Readonly True

Type int

`callback`

Callback which should be called on IO events.

`callback`(*Poll-Handle*, *Status-Code*, *Event-Mask*)

Readonly False

Type (uv.Poll, uv.StatusCode, int) -> None

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

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

Raises

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

Parameters

- **events** (*int*) – bitmask of events which should be polled for
- **callback** (*(uv.Poll, uv.StatusCode, int) -> None*) – callback which should be called on IO events

stop ()

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

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

class uv.**PollEvent**

Poll event types enumeration.

READABLE = None

File descriptor is readable.

Type int

WRITABLE = None

File descriptor is writable.

Type int

1.7 `Prepare` – poll handle

class uv.**Prepare** (*loop=None*, *callback=None*)

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

Raises **`uv.UVError`** – error during the initialization of the handle

Parameters

- **loop** (*Loop*) – event loop which should be used for the handle
- **callback** (*(uv.Prepare) -> None*) – callback which should be called right before polling for IO

callback

Callback which should be called before polling for IO.

callback (*Prepare-Handle*)

Readonly False

Type (uv.Prepare) -> None

start (*callback=None*)
Starts the handle.

Raises

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

Parameters **callback** (*(uv.Prepare) -> None*) – callback which should be called before polling for IO

stop ()
Stops the handle, the callback will no longer be called.

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

1.8 Signal – signal handle

class uv.**Signal** (*loop=None*, *callback=None*)
Signal handles implement Unix style signal handling on a per-event loop bases. 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 during the initialization of the handle

Parameters

- **loop** (*Loop*) – event loop which should be used for the handle
- **callback** (*(uv.Signal, int) -> None*) – callback which should be called on signal delivery

callback
Callback which should be called on signal delivery.

callback (*Signal-Handle*, *Signal-Number*)

Readonly False

Type (uv.Signal, int) -> None

signum
Signal being monitored by this handle.

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

Readonly True

Return type int

start (*signum*, *callback=None*)
Starts the handle.

Raises

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

Parameters

- **signum** (*int*) – signal number which should be monitored
- **callback** (*(uv.Signal) -> None*) – callback which should be called on signal delivery

stop()

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

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

class uv.**Signals**

Generic 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 int

SIGBREAK = None

Is delivered when the user presses CTRL+BREAK.

Type int

SIGHUP = None

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

Type int

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 int

1.9 `Timer` – timer handle

class uv.**Timer**(*loop=None*, *callback=None*)

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

Raises ***uv.UVError*** – error during the initialization of the handle

Parameters

- **loop** (*Loop*) – event loop which should be used for the handle
- **callback** (*(uv.Timer) -> None*) – callback which should be called on timeout

callback

Callback which should be called on timeout.

callback(*Timer-Handle*)

Readonly False

Type (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.HandleClosedError`*** – 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.UVError`* with `uv.StatusCode.EINVAL`.

Raises

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

`start`(*timeout*, *callback=None*, *repeat=0*)

Starts 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.HandleClosedError`*** – handle has already been closed or is closing

Parameters

- **`timeout`** (*`int`*) – timeout which should be used (in milliseconds)
- **`callback`** (*`(uv.Timer) -> None`*) – callback which should be called on timeout
- **`repeat`** (*`int`*) – repeat interval which should be set (in milliseconds)

`stop``()`

Stops 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

Index