blocklandjs Documentation

Release v8.1.2

metario

Mar 07, 2018

Contents:

CHAPTER 1

Introduction

This is a JavaScript implementation for the game Blockland, using the high performance V8 Javascript engine.

CHAPTER 2

How to install

Download latest release from GitHub, place in modules/ folder, and you're ready to go!

2.1 JavaScript documentation

2.1.1 console

`console.`**`print`**(*text*)
`console.`**`log`**(*text*)
`console.`**`warn`**(*text*)
`console.`**`error`**(*text*)
Print out a string.

> **Warning:** Ensure that the string is < 4096 characters, or else it will not print out the intended string.

Arguments

- **`text`** (*`string`*) – Text to be printed.

2.1.2 globals

`print`(*text*)
Print out text to the console. Same function as `console.log`

> **Warning:** Ensure that the string is < 4096 characters, or else it will not print out the intended string.

Arguments

- **text** (*string*) – String to print out to console.

version()
Get the current version of V8 running.

Returns A string containing the current version of V8 running.

immediateMode(*enable*)
Enable or disable "immediate mode". Used when extreme precision for `uv.misc.hrtime()` is needed.

> **Warning:** CPU usage will be extremely high if immediateMode is enabled.

Arguments

- **enable** (*bool*) – Enable/disable immediate mode.

load(*file*)
Load and evaluate a JavaScript file.

Arguments

- **file** (*string*) – Path to the JavaScript file. The path can be relative to the current working directory.

2.1.3 LibUV

Introduction

This is an implementation of the popular libuv library, which is used in node.js. It provides the event loop (timeouts, timers, etc), as well as a full tcp stack.

Classes

fs

`uv.fs.`**close**(*fd*)
See the manpages for close(2)

Arguments

- **fd** (*int*) – A file descriptor that is open.

Throws If unable to close the file descriptor/the file descriptor does not exist.

`uv.fs.`**open**(*path*, *flags*, *callback*)
See the manpages for open(2)

Arguments

- **path** (*string*) – The path to the file. May be relative to the current working directory.
- **flags** (*string*) – Flags to open the file with.
- **callback** (*function(fd)*) – The callback that will recieve the file descriptor of the newly opened file.

uv.fs.**read**(*fd*, *size*, *offset*, *callback*)
See the manpages for preadv(2)

Arguments

- **fd** (*int*) – The opened file-descriptor.
- **size** (*int*) – The amount of bytes to read from the file.
- **offset** (*int*) – The offset to read from.
- **callback** (*function(ArrayBuffer)*) – The callback function which will take an ArrayBuffer as it's first argument.

uv.fs.**unlink**(*fd*, *callback*)
See the manpages for unlink(2)

Arguments

- **fd** (*int*) – The opened file-descriptor
- **callback** (*function(bool)*) – A callback that will accept a boolean as it's first argument, indicating the status of the request.

uv.fs.**write**(*fd*, *array*, *offset*, *callback*)
See the manpages for pwritev(2)

Arguments

- **fd** (*int*) – The opened file-descriptor.
- **array** (*Uint8Array*) – The bytes to write out to the file.
- **offset** (*int*) – The offset for writing to the file.
- **callback** (*function(bool)*) – A callback that will accept a boolean as it's first argument, indicating the status of the request.

uv.fs.**mkdir**(*path*, *callback*)
See the manpages for mkdir(2)

Arguments

- **path** (*string*) – The path to create.
- **callback** (*function(bool)*) – The callback which will recieve the status of the request.

uv.fs.**mkdtemp**(*path*, *callback*)
See the manpages for mkdtemp(3)

uv.fs.**rmdir**(*path*, *callback*)
See the manpages for rmdir(2)

uv.fs.**scandir**(*path*, *callback*)
See the manpages for scandir(3)

After this function finishes execution, the callback will recieve a function which you can repeatedly call to get the next file.

uv.fs.**stat**(*path*, *callback*)
See the manpages for stat(2)

uv.fs.**fstat**(*fd*, *callback*)
See the manpages for fstat(2)

`uv.fs.lstat` (*path*, *callback*)
See the manpages for lstat(2)

`uv.fs.rename` (*path*, *new_path*, *callback*)
See the manpages for rename(2)

`uv.fs.fsync` (*fd*, *callback*)
See the manpages for fsync(2)

`uv.fs.fdatasync` (*fd*, *callback*)
See the manpages for fdatasync(2)

`uv.fs.ftruncate` (*fd*, *offset*, *callback*)
See the manpages for ftruncate(2)

`uv.fs.sendfile` (*fd*, *out_fd*, *offset*, *length*, *callback*)
See the manpages for sendfile(2)

`uv.fs.access` (*path*, *flags*, *callback*)
See the manpages for access(2)

`uv.fs.chmod` (*path*, *flags*, *callback*)
See the manpages for chmod(2)

`uv.fs.fchmod` (*fd*, *flags*, *callback*)
See the manpages for fchmod(2)

`uv.fs.utime` (*path*, *atime*, *mtime*, *callback*)
See the manpages for utime(2)

`uv.fs.futime` (*fd*, *atime*, *mtime*, *callback*)
See the manpages for futime(2)

`uv.fs.link` (*path*, *new_path*, *callback*)
See the manpages for link(2)

`uv.fs.symlink` (*path*, *new_path*, *flags*, *callback*)
See the manpages for symlink(2)

`uv.fs.readlink` (*path*, *callback*)
See the manpages for readlink(2)

`uv.fs.chown` (*path*, *uid*, *gid*, *callback*)
See the manpages for chown(2)

`uv.fs.fchown` (*fd*, *uid*, *gid*, *callback*)
See the manpages for fchown(2)

misc

Misc functions for a tinkering user.

`uv.misc.version`()
Get the current version of LibUV (as a integer)

Returns A integer representing the current libuv version.

`uv.misc.version_string`()
Get the current version of LibUV (as a string)

Returns A string representing the current libuv version.

uv.misc.**get_process_title**()
Get the current process title.

Returns A string representing the current process title.

uv.misc.**set_process_title**(*name*)
Set the process title of the executable.

Arguments

- **name** (*string*) – The new process title.

uv.misc.**resident_set_memory**()
Get the resident set size for the current process.

Returns A integer representing the resident set size

uv.misc.**uptime**()
Get the current system uptime

Returns A integer representing the current system uptime (in milliseconds)

uv.misc.**getrusage**()
Get the current resource usage of the system.

Returns An object containing various fields relating to resource usage.

uv.misc.**cpu_info**()
Get info on the processors on the host system.

object.**model**
Model of CPU.

object.**speed**
Speed of CPU.

object.**times**

Times for the CPU.

object.times.**user**
"User" times for the CPU.

object.times.**nice**
"Nice" times for the CPU.

object.times.**sys**
"Sys" times for the CPU.

object.times.**idle**
"Idle" times for the CPU.

object.times.**irq**
"IRQ" times for the CPU.

Returns An array containing objects containing information about the processors in the host system.

uv.misc.**interface_addresses**()
Get the address, family, and port of the network interfaces on the host system.

object.**name**
Name of the network interface.

object.**internal**
Boolean representing if the object is an internal interface or not.

`object.`**address**
String representing the IPv4/IPv6 address of the interface.

`object.`**protocol**
String representing the protocol of the interface.

Returns An array containing objects containing information about the network interfaces in the host.

`uv.misc.`**loadavg**`()`
Get the load average of the system.

Returns An array containing three numbers representing the load averages of the system.

`uv.misc.`**exepath**`()`
Get the path to the running executable.

Returns A string corresponding to the path of the current running executable.

`uv.misc.`**cwd**`()`
Get the current working directory.

Returns A string corresponding to the current working directory.

`uv.misc.`**os_homedir**`()`
Get the home directory of the current logged in user.

Returns A string corresponding to the home directory of the current user.

`uv.misc.`**chdir**`(`*path*`)`
Change the current working directory to the path.

> **Warning:** Be sure to change the current working path to the Blockland/ directory, or else Torque will be fucked.

Arguments

- **path** (*string*) – The path to change the current working directory to.

`uv.misc.`**get_total_memory**`()`
Get the current memory of the system.

Returns A number representing the total memory, in kilobytes.

`uv.misc.`**hrtime**`()`
Get the current high-resolution timestamp of the system.

Returns A number representing the current high-resolution timestamp, in nanoseconds.

`uv.misc.`**now**`()`
Get a value corresponding to some value stored in the timer loop. This is relative to some random point in time in the past.

Returns A number representing a timestamp.

pipe

class `uv.`**pipe**`(`*ipc*`)`
Construct a new named pipe.

Arguments

- **ipc** (*bool*) – A toggle for if this pipe is going to be used for IPC.

Note: The `Stream` object methods also apply here.

`uv.pipe.`**bind**(*path*)
Bind the pipe to a named pipe/path.

Arguments

- **path** (*string*) – Path, or the named pipe.

`uv.pipe.`**open**(*fd*)
Open the pipe on an existing file descriptor or HANDLE.

Arguments

- **fd** (*int*) – The file descriptor to open the `Pipe` on.

Throws If unable to open the pipe on said file descriptor.

`uv.pipe.`**connect**(*pipe*)
Connect to the named pipe/socket.

Arguments

- **pipe** (*string*) – The named pipe to connect to.

`uv.pipe.`**getsockname**()
Get the name of the socket.

Returns A string representing the name of the socket/pipe.

`uv.pipe.`**getpeername**()
Get the name of the named pipe that the pipe is connected to.

Returns A string representing the name of the peer.

`uv.pipe.`**pending_instances**(*count*)
Set the number of pending pipe instance handles when the pipe server is waiting for connections.

Arguments

- **count** (*int*) – Number of pending pipe instance handles.

`uv.pipe.`**pending_count**()
Get pending count of pipe. Used to transmit types.

Returns An integer representing the pending count of handles.

`uv.pipe.`**pending_type**()
Get pending type of handle being transmitted.

Returns An integer representing the type to be transmitted.

`uv.pipe.`**chmod**(*flags*)
Alter pipe permissions.

Arguments

- **flags** (*int*) – The flags that should now be set to the pipe.

stream

`uv.stream.`**`shutdown`**`()`
Shutdown a stream that is connected, disabling the write side. The shutdown callback will be called directly after.

> **Warning:** Ensure that you do not close the stream at the same time that you shutdown. Close after the shutdown callback has been called.

`uv.stream.`**`listen`**`(`*backlog*`)`
Set a `Stream` object to listen.

Arguments

- **`backlog`** (*`int`*) – The number of connections the kernel may queue.

`uv.stream.`**`accept`**`()`
Accept a new connection on a `Stream`.

Returns An object corresponding to the original type of the `Stream` that this function was called on.

`uv.stream.`**`read_start`**`()`
Start reading from the `Stream`.

Throws If unable to start reading from the Stream (disconnected, etc.)

`uv.stream.`**`read_stop`**`()`
Stop reading data from the `Stream`.

Note: The 'data' callback will not be called beyond this point, if new data is recieved.

`uv.stream.`**`write`**`(`*buffer*`)`
Write data to the `Stream`.

Arguments

- **`buffer`** (*`Uint8Array`*) – A Uint8Array containing the raw data you want to write to the `Stream`.

Throws If unable to write to the `Stream`.

`uv.stream.`**`on`**`(`*event*, *callback*`)`
Set a callback for an event occuring. Possible events are

Event	Callback schema
data	function(len, buffer)
connection	function(status)
write	function(status)
connect	function(status)
close	function()
shutdown	function(status)

Arguments

- **`event`** (*`string`*) – A string indicating the event that you want to listen for.

- **callback** (*function*) – A function that should be called upon these events occuring.

uv.stream.**is_writable**()
Returns a value indicating if the Stream can be written to.

Returns A boolean indicating if you can write to the Stream.

uv.stream.**is_readable**()
Returns a value indicating if the Stream can be read from.

Returns A boolean indicating if you can read from the Stream

uv.stream.**set_blocking**(*enable*)
Set if the Stream should block.

Arguments

- **enable** (*bool*) – Whether the Stream should block or not.

uv.stream.**close**()
Close the Stream.

Note: If you call this while there are pending events, there is a very high likelyhood that the game will crash. Ensure that you shutdown the Stream, before closing it.

Throws If unable to close the Stream.

tcp

class uv.**tcp**()
Construct a new Tcp object, inheriting from the Stream object.

Note: Stream object methods apply here.

---

uv.tcp.**open**(*fd*)
Open an existing file descriptor/SOCKET as a TCP handle.

Arguments

- **fd** (*number*) – existing file descriptor or socket.

Throws If unable to open

uv.tcp.**nodelay**(*enable*)
Enable TCP_NODELAY, which disables Nagle's algorithm.

Arguments

- **enable** (*bool*) – enable or disable nodelay

Throws If unable to set nodelay

uv.tcp.**simultaneous_accepts**(*enable*)
Enable / disable simultaneous asynchronous accept requests

Arguments

- **enable** (*bool*) – enable or disable simultaneous accepts for object.

Throws If unable to set simultaneous_accepts

`uv.tcp.`**`keepalive`**(*enable*, *delay*)
Enable / disable TCP keepalive.

Arguments

- **`enable`** (*`bool`*) – enable or disable keepalive
- **`delay`** (*`int`*) – initial delay for keepalive in seconds.

Throws If unable to set keepalive

`uv.tcp.`**`bind`**(*host*, *port*)
Bind to a specified host/port.

Arguments

- **`host`** (*`string`*) – IP address to bind to.
- **`port`** (*`int`*) – Port to bind to.

Throws If unable to bind to host/port

`uv.tcp.`**`getpeername`**()
Get the address of the client/peer connected to the handle.

Returns An object containing the family, port, and IP of the peer.

Throws If unable to get peername

`uv.tcp.`**`getsockname`**()
Get the address of the socket (us)

Returns An object containing the family, port, and IP of the socket.

Throws If unable to get socket name.

`uv.tcp.`**`connect`**(*host*, *port*)
Connect a socket to a host, and port.

Arguments

- **`host`** (*`string`*) – IP address to connect to.
- **`port`** (*`int`*) – Port to connect to.

Throws If unable to connect to host/port.

timer

`class` `uv.`**`timer`**()
A timer class, providing similar functionality of the default `setTimeout`, and `setRepeat`

`uv.timer.`**`start`**(*timeout*, *repeat*, *callback*, *arguments*)
Start the timer.

Arguments

- **`timeout`** (*`int`*) – The time that the timer should wait before first calling the callback (in milliseconds).
- **`repeat`** (*`int`*) – The time between repeat calls of the timer (in milliseconds).
- **`callback`** (*`function`*) – The callback that should be called every time that this timer ticks.

- **arguments** (*Array*) – arguments to pass to the callback function

`uv.timer.`**`stop`**`()`
Stop the timer.

Throws If the timer is not running.

`uv.timer.`**`again`**`()`
Start the timer with the parameters specified before, after a `stop()` call has been made.

Throws If this is the timer's first go.

`uv.timer.`**`set_repeat`**`(`*repeat*`)`
Set the time between repeat calls of this timer.

Arguments

- **repeat** (*int*) – The time between repeat calls of the timer (in milliseconds).

Throws If the timer has not been started.

`uv.timer.`**`get_repeat`**`()`
Get the time between repeat calls of this timer.

Returns An integer representing the time between repeat calls of this timer (in milliseconds).

tty

`class` `uv.`**`tty`**`(`*fd*, *readable*`)`
A class representing a TTY stream. Inherits from the `Stream` object.

Arguments

- **fd** (*int*) – A file descriptor.
- **readable** (*int*) – Whether the TTY is readable or not.

Throws If construction failed.

`uv.tty.mode.`**`set`**`()`

> **Warning:** Unfinished.

`uv.tty.mode.`**`reset`**`()`

> **Warning:** Unfinished.

`uv.tty.`**`get_winsize`**`()`

> **Warning:** Unfinished.

2.1.4 sqlite

class sqlite()
Construct a new SQLite database worker.

sqlite.**open**(*database*)
Open a new database.

Throws If unable to open the database.

sqlite.**exec**(*query*[, *callback*])
Execute a SQL query. The callback will be called with the first argument as an array, holding the results.

Throws If a SQLite error is occured, or the database is not opened.

sqlite.**close**()
Close an opened database.

Throws If unable to close the database, or there is not a database currently opened.

2.1.5 TorqueScript bridge

The TorqueScript bridge is a way to communicate between JavaScript and TorqueScript. It is purely meant as a compatibility layer.

ts.**setVariable**(*name*, *value*)
This will set a variable within the TorqueScript global scope to value.

> **Warning:** Ensure that the value is coercible to a string/a string!

Arguments

- **name** (*string*) – The name of the variable to set value to.
- **value** (*string*) – The value of the variable.

ts.**getVariable**(*name*)
This will return a variable corresponding to a TorqueScript global variable.

> **Note:** This will ALWAYS return a string, due to how TorqueScript handles internal types.

Arguments

- **name** (*string*) – The name of the variable to get the value of.

Returns A string corresponding to the value held by the variable.

ts.**linkClass**(*name*)
Returns a constructor for a TorqueScript class.

> **Note:** Be sure to register any object created by this constructor. If you fail to register an object, no method call will work, as the object has no ID.

> **Warning:** Do not link a datablock class. This will crash the game. I strongly advise AGAINST doing this.

Arguments

- **name** (*string*) – The name of the class corresponding to a TorqueScript class.

Returns A function that works as a constructor for the TorqueScript type specified.

`ts.`**`registerObject`**`(`*object*`)`
Register an object with Torque, allowing for method functions to be called on it, and giving it an object id.

Arguments

- **object** (*object*) – The object to be registered with Torque.

`ts.`**`func`**`(`*name*`)`
Get a JavaScript function corresponding to the TorqueScript function.

Arguments

- **name** (*string*) – The name of the function in the global TorqueScript namespace.

Returns A JavaScript function that corresponds to the TorqueScript function

`ts.`**`obj`**`(`*name*`)`
`ts.`**`obj`**`(`*id*`)`
Get an object referring to the TorqueScript object given by the name/id

Arguments

- **id** (*int*) – The ID corresponding to the TorqueScript object.
- **name** (*string*) – The name corresponding to the TorqueScript object.

Returns An object referring to the TorqueScript object given by the id/name.

`ts.`**`switchToTS`**`()`
Change your in-game console to use TorqueScript instead of JavaScript.

`ts.`**`expose`**`(`*info*, *function*`)`

`info.`**`class`**
The class that the function should be registered to. Optional.

`info.`**`name`**
The name that the function should be registered as.

`info.`**`description`**
The description that the function should have. Optional.

Arguments

- **info** (*object*) – An object containing all of the attributes listed above.
- **function** (*function*) – A function that should be called every time the TorqueScript callback is called.

Exposes a JavaScript function to TorqueScript.

`ts.SimSet.`**`getObject`**`(`*SimSet*, *id*`)`
Get an object inside of a SimSet.

Arguments

- **SimSet** (*object*) – An object referring to a TorqueScript SimSet.
- **id** (*int*) – The integer referring to the object's position within the SimSet.

Returns An object that is found at the index given, inside of the SimSet.

`ts.SimSet.`**getCount** (*SimSet*)
Get the count of all the objects inside the SimSet.

Arguments

- **SimSet** (*object*) – An object referring to a TorqueScript SimSet.

Returns An integer representing the number of objects inside of the SimSet.

2.2 Working with BlocklandJS

Index