
USB Library for Node.JS



Node.JS library for communicating with USB devices.

This is a refactoring / rewrite of Christopher Klein's node-usb.

Prerequisites

Node.js \geq v10.20.0, which includes [npm](#).

Windows

On Windows, if you get `LIBUSB_ERROR_NOT_SUPPORTED` when attempting to open your device, it's possible your device doesn't have a WinUSB driver for libusb to use.

You can install one using Zadig or another approach is to use the UsbDK Backend of libusb by immediately calling `usb.useUsbDkBackend()`.

Note that you cannot use multiple drivers on Windows as they get exclusive access to the device. So if you want to switch between drivers (e.g. using a printer with this software or the system), you will need to uninstall/install drivers as required.

For further info, check How to use libusb on Windows in the libusb's wiki.

Linux

On Linux, you'll need libudev to build libusb if a prebuild is not available. On Ubuntu/Debian:

```
1 sudo apt-get install build-essential libudev-dev
```

Troubleshooting

For libusb issues, please refer to the FAQ at <https://github.com/libusb/libusb/wiki/FAQ>

Installation

Native modules are bundled using `prebuildify`, so installation should be as simple as installing the package.

With `npm`:

```
1 npm install usb
```

With `yarn`:

```
1 yarn add usb
```

Note: the library is now written in `TypeScript`, so a separate types file is not longer required to be installed (e.g. don't install `@types/usb`).

License

MIT

Note that the compiled Node extension includes `libusb`, and is thus subject to the LGPL.

Limitations

Does not support:

- Configurations other than the default one
- Isochronous transfers

Getting Started

Use the following examples to kickstart your development. Once you have a desired device, use the APIs below to interact with it.

Migrating to v2.0.0

The legacy API exists on an object called `usb` on the main import and the convenience functions exist as top level objects.

To use `v2.0.0` simply update your import statements and the function calls;

```
1 // import * as usb from 'usb';
2 // const devices: usb.Device[] = usb.getDeviceList();
3
4 import { usb, getDeviceList } from 'usb';
5 const devices: usb.Device[] = getDeviceList();
```

List all legacy devices

```
1 import { getDeviceList } from 'usb';
2
3 const devices = getDeviceList();
4
5 for (const device of devices) {
6     console.log(device); // Legacy device
7 }
```

Find legacy device by vid/pid

```
1 import { findByIds } from 'usb';
2
3 const device = findByIds(0x59e3, 0x0a23);
4
5 if (device) {
6     console.log(device); // Legacy device
7 }
```

Find legacy device by SerialNumber

```
1 import { findBySerialNumber } from 'usb';
2
3 (async () => {
4     // Uses a blocking call, so is async
5     const device = await findBySerialNumber('TEST_DEVICE');
6
7     if (device) {
8         console.log(device); // Legacy device
9     }
10 })();
```

Turn legacy Device into WebUSB compatible device

```
1 import { findBySerialNumber, WebUSBDevice } from 'usb';
2
3 (async () => {
4     // Uses a blocking call, so is async
5     const device = await findBySerialNumber('TEST_DEVICE');
6
7     // Uses blocking calls, so is async
8     const webDevice = await WebUSBDevice.createInstance(device);
9
10    if (webDevice) {
11        console.log(webDevice); // WebUSB device
12    }
13 })();
```

Use WebUSB approach to find a device

```
1 import { webusb } from 'usb';
2
3 (async () => {
4     // Returns first matching device
5     const device = await webusb.requestDevice({
6         filters: [{}]}
7     });
8
9     console.log(device); // WebUSB device
10 })();
```

Use WebUSB approach to find a device with custom selection method

```
1 import { WebUSB } from 'usb';
2
3 (async () => {
4     const customWebUSB = new WebUSB({
5         // This function can return a promise which allows a UI to be
5         // displayed if required
6         devicesFound: devices => devices.find(device => device.
6             serialNumber === 'TEST_DEVICE')
7     });
8
9     // Returns device based on injected 'devicesFound' function
10    const device = await customWebUSB.requestDevice({
11        filters: [{}]}
12    );
13
14    console.log(device); // WebUSB device
15 })();
```

Use WebUSB approach to list authorised devices

```
1 import { webusb } from 'usb';
2
3 (async () => {
4     // The default webusb instance follows the WebUSB spec and only
4     // returns authorised devices
5     const devices = await webusb.getDevices();
6
7     for (const device of devices) {
8         console.log(device); // WebUSB device
9     }
10 })();
```

Use WebUSB approach to list all devices

```
1 import { WebUSB } from 'usb';
2
3 (async () => {
4     const customWebUSB = new WebUSB({
5         // Bypass checking for authorised devices
6         allowAllDevices: true
7     });
8
9     // Uses blocking calls, so is async
10    const devices = await customWebUSB.getDevices();
11
12    for (const device of devices) {
13        console.log(device); // WebUSB device
14    }
15 })();
```

Electron

Please refer to the maintained example for using `node-usb` in electron:

<https://github.com/node-usb/node-usb-example-electron>

APIs

Since `v2.0.0`, the `node-usb` library supports two APIs:

- `WebUSB` which follows the WebUSB Specification (recommended)

-
- **Legacy API** which retains the previous ‘non-blocking’ API

Convenience methods also exist to easily list or find devices as well as convert between a legacy `usb.Device` device and WebUSB device.

Full auto-generated API documentation can be seen here:

<https://node-usb.github.io/node-usb/>

Convenience Functions

getDeviceList()

Return a list of legacy `Device` objects for the USB devices attached to the system.

findByIds(vid, pid)

Convenience method to get the first legacy device with the specified VID and PID, or `undefined` if no such device is present.

findBySerialNumber(serialNumber)

Convenience method to get a promise of the legacy device with the specified serial number, or `undefined` if no such device is present.

getWebUsb()

Return the `navigator.usb` instance if it exists, otherwise a `webusb` instance.

WebUSBDevice

WebUSB Device class for wrapping a legacy Device into a WebUSB device

WebUSBDevice.createInstance(device) Convenience method to return a promise of a WebUSB device based on a legacy device

WebUSB

Please refer to the WebUSB specification which be found here:

<https://wicg.github.io/webusb/>

Implementation Status

USB

- ☒ `getDevices()`
- ☒ `requestDevice()`

USBDevice

- ☒ `usbVersionMajor`
- ☒ `usbVersionMinor`
- ☒ `usbVersionSubminor`
- ☒ `deviceClass`
- ☒ `deviceSubclass`
- ☒ `deviceProtocol`
- ☒ `vendorId`
- ☒ `productId`
- ☒ `deviceVersionMajor`
- ☒ `deviceVersionMinor`
- ☒ `deviceVersionSubminor`
- ☒ `manufacturerName`
- ☒ `productName`
- ☒ `serialNumber`
- ☒ `configuration`
- ☒ `configurations`
- ☒ `opened`
- ☒ `open()`
- ☒ `close()`
- ☒ `selectConfiguration()`
- ☒ `claimInterface()`
- ☒ `releaseInterface()`
- ☒ `selectAlternateInterface()`
- ☒ `controlTransferIn()`

-
- ☒ `controlTransferOut()` - `bytesWritten` always equals the initial buffer length
 - ☒ `transferIn()`
 - ☒ `transferOut()` - `bytesWritten` always equals the initial buffer length
 - ☒ `clearHalt()`
 - ☒ `reset()`
 - ☐ `isochronousTransferIn()`
 - ☐ `isochronousTransferOut()`
 - ☐ `forget()`

Events

- ☒ `connect`
- ☒ `disconnect`

Legacy API

usb

Legacy usb object.

usb.LIBUSB_* Constant properties from libusb

usb.getDeviceList() Return a list of legacy `Device` objects for the USB devices attached to the system.

usb.pollHotplug Force polling loop for hotplug events.

usb.setDebugLevel(level : int) Set the libusb debug level (between 0 and 4)

usb.useUsbDkBackend() On Windows, use the USBDK backend of libusb instead of WinUSB

Device

Represents a USB device.

.busNumber Integer USB device number

.deviceAddress Integer USB device address

.portNumbers Array containing the USB device port numbers, or `undefined` if not supported on this platform.

.deviceDescriptor Object with properties for the fields of the device descriptor:

- bLength
- bDescriptorType
- bcdUSB
- bDeviceClass
- bDeviceSubClass
- bDeviceProtocol
- bMaxPacketSize0
- idVendor
- idProduct
- bcdDevice
- iManufacturer
- iProduct
- iSerialNumber
- bNumConfigurations

.configDescriptor Object with properties for the fields of the configuration descriptor:

- bLength
- bDescriptorType
- wTotalLength
- bNumInterfaces
- bConfigurationValue
- iConfiguration
- bmAttributes
- bMaxPower
- extra (Buffer containing any extra data or additional descriptors)

.allConfigDescriptors Contains all config descriptors of the device (same structure as .configDescriptor above)

.parent Contains the parent of the device, such as a hub. If there is no parent this property is set to **null**.

.open() Open the device. All methods below require the device to be open before use.

.close() Close the device.

.controlTransfer(bmRequestType, bRequest, wValue, wIndex, data_or_length, callback(error, data)) Perform a control transfer with `libusb_control_transfer`.

Parameter `data_or_length` can be a integer length for an IN transfer, or a Buffer for an out transfer. The type must match the direction specified in the MSB of `bmRequestType`.

The `data` parameter of the callback is always undefined for OUT transfers, or will be passed a Buffer for IN transfers.

A package is available to calculate `bmRequestType` if needed.

.setConfiguration(id, callback(error)) Set the device configuration to something other than the default (0). To use this, first call `.open(false)` (which tells it not to auto configure), then before claiming an interface, call this method.

.getStringDescriptor(index, callback(error, data)) Perform a control transfer to retrieve a string descriptor

.getBosDescriptor(callback(error, bosDescriptor)) Perform a control transfer to retrieve an object with properties for the fields of the Binary Object Store descriptor:

- `bLength`
- `bDescriptorType`
- `wTotalLength`
- `bNumDeviceCaps`

.getCapabilities(callback(error, capabilities)) Retrieve a list of Capability objects for the Binary Object Store capabilities of the device.

.interface(interface) Return the interface with the specified interface number.

.interfaces List of Interface objects for the interfaces of the default configuration of the device.

.timeout Timeout in milliseconds to use for control transfers.

.reset(callback(error)) Performs a reset of the device. Callback is called when complete.

Interface

.endpoint(address) Return the InEndpoint or OutEndpoint with the specified address.

.endpoints List of endpoints on this interface: InEndpoint and OutEndpoint objects.

.interface Integer interface number.

.altSetting Integer alternate setting number.

.setAltSetting(altSetting, callback(error)) Sets the alternate setting. It updates the **interface** **.endpoints** array to reflect the endpoints found in the alternate setting.

.claim() Claims the interface. This method must be called before using any endpoints of this interface.

.release([closeEndpoints], callback(error)) Releases the interface and resets the alternate setting. Calls callback when complete.

It is an error to release an interface with pending transfers. If the optional closeEndpoints parameter is true, any active endpoint streams are stopped (see **Endpoint.stopStream**), and the interface is released after the stream transfers are cancelled. Transfers submitted individually with **Endpoint.transfer** are not affected by this parameter.

.isKernelDriverActive() Returns **false** if a kernel driver is not active; **true** if active.

.detachKernelDriver() Detaches the kernel driver from the interface.

.attachKernelDriver() Re-attaches the kernel driver for the interface.

.descriptor Object with fields from the interface descriptor – see libusb documentation or USB spec.

- bLength
- bDescriptorType
- bInterfaceNumber
- bAlternateSetting
- bNumEndpoints
- bInterfaceClass
- bInterfaceSubClass
- bInterfaceProtocol
- iInterface
- extra (Buffer containing any extra data or additional descriptors)

Capability

.type Integer capability type.

.data Buffer capability data.

.descriptor Object with fields from the capability descriptor – see libusb documentation or USB spec.

- bLength
- bDescriptorType
- bDevCapabilityType

Endpoint

Common base for InEndpoint and OutEndpoint, see below.

.direction Endpoint direction: "in" or "out".

.transferType Endpoint type: `usb.LIBUSB_TRANSFER_TYPE_BULK`, `usb.LIBUSB_TRANSFER_TYPE_INTERRUPT`, or `usb.LIBUSB_TRANSFER_TYPE_ISOCHRONOUS`.

.descriptor Object with fields from the endpoint descriptor – see libusb documentation or USB spec.

- `bLength`
- `bDescriptorType`
- `bEndpointAddress`
- `bmAttributes`
- `wMaxPacketSize`
- `bInterval`
- `bRefresh`
- `bSynchAddress`
- `extra` (Buffer containing any extra data or additional descriptors)

.timeout Sets the timeout in milliseconds for transfers on this endpoint. The default, 0, is infinite timeout.

.clearHalt(callback(error)) Clear the halt/stall condition for this endpoint.

InEndpoint

Endpoints in the IN direction (device->PC) have this type.

.transfer(length, callback(error, data)) Perform a transfer to read data from the endpoint.

If `length` is greater than `maxPacketSize`, libusb will automatically split the transfer in multiple packets, and you will receive one callback with all data once all packets are complete.

this in the callback is the `InEndpoint` object.

.startPoll(nTransfers=3, transferSize=maxPacketSize) Start polling the endpoint.

The library will keep `nTransfers` transfers of size `transferSize` pending in the kernel at all times to ensure continuous data flow. This is handled by the libusb event thread, so it continues even if the Node v8 thread is busy. The `data` and `error` events are emitted as transfers complete.

.stopPoll(cb) Stop polling.

Further data may still be received. The **end** event is emitted and the callback is called once all transfers have completed or canceled.

Event: data(data : Buffer) Emitted with data received by the polling transfers

Event: error(error) Emitted when polling encounters an error. All in flight transfers will be automatically canceled and no further polling will be done. You have to wait for the **end** event before you can start polling again.

Event: end Emitted when polling has been canceled

OutEndpoint

Endpoints in the OUT direction (PC->device) have this type.

.transfer(data, callback(error)) Perform a transfer to write **data** to the endpoint.

If length is greater than `maxPacketSize`, libusb will automatically split the transfer in multiple packets, and you will receive one callback once all packets are complete.

this in the callback is the OutEndpoint object.

Event: error(error) Emitted when the stream encounters an error.

Event: end Emitted when the stream has been stopped and all pending requests have been completed.

UsbDetection

usb.on('attach', function(device) { ... }); Attaches a callback to plugging in a **device**.

usb.on('detach', function(device) { ... }); Attaches a callback to unplugging a **device**.

usb.refHotplugEvents(); Restore (re-reference) the hotplug events unreferenced by `usb.unrefHotplugEvents()`

usb.unrefHotplugEvents(); Listening to events will prevent the process to exit. By calling this function, hotplug events will be unreferenced by the event loop, allowing the process to exit even when listening for the `attach` and `detach` events.

Migrating from node-usb-detection

If you have been referred here by `node-usb-detection`, the following may be helpful for you to update your existing code.

usbDetect.startMonitoring() & usbDetect.stopMonitoring()

There is no direct equivalent to these methods. This is handled automatically for you when you add and remove event listeners.

You may find `usb.unrefHotplugEvents()` useful as it is intended to help with exit conditions.

usbDetect.find()

You can instead use `usb.getDeviceList()`. Be aware that this will do an enumeration to find all of the devices, and not look at a cache of the known devices. If you call it too often it may have a performance impact.

To find a specific device by vid and pid, call `usb.findByIds`. e.g. `usb.findByIds(0x12, 0x34)`.

usbDetect.on('add', function(device) { ... })

These should be changed to `usb.on('attach', function(device) { ... })`.

There is no equivalent to filter based on the vid or pid, instead you should do a check inside the callback you provide.

The contents of the device object has also changed.

usbDetect.on('remove', function(device) { ... })

These should be changed to `usb.on('detach', function(device) { ... })`.

There is no equivalent to filter based on the vid or pid, instead you should do a check inside the callback you provide.

The contents of the device object has also changed.

usbDetect.on('change', function(device) { ... })

There is no direct equivalent to this. Instead you can listen to both `attach` and `detach` to get the same behaviour.

Development

The library is based on native bindings wrapping the libusb library.

Setup

Libusb is included as a submodule, clone this repository and then the submodule as follows:

```
1 git clone https://github.com/node-usb/node-usb
2 cd node-usb
3 git submodule update --init
```

Building

The package uses `prebuildify` to generate the native binaries using an `install` script and `TypeScript` for the binding code using the `compile` script. The package can be built as follows:

```
1 yarn
2 yarn compile
```

The native binaries can be rebuilt with:

```
1 yarn rebuild
```

Note: On Linux, you'll need libudev to build libusb. On Ubuntu/Debian:

```
1 sudo apt-get install build-essential libudev-dev
```

Testing

To execute the unit tests, Run:

```
1 yarn test
```

Some tests require an attached STM32F103 Microprocessor USB device with specific firmware.

```
1 yarn full-test  
2 yarn valgrind
```

Releasing

Please refer to the Wiki for release instructions.