

Access USB Devices on the Web



By François Beaufort

Dives into Chromium source code

If I said plain and simple "USB", there is a good chance that you will immediately think of keyboards, mice, audio, video and storage devices. You're right but you'll find other kinds of Universal Serial Bus (USB) devices out there.

These non-standardized USB devices require hardware vendors to write native drivers and SDKs in order for you (the developer) to take advantage of them. Sadly this native code has historically prevented these devices from being used by the Web. And that's one of the reasons the WebUSB API has been created: to provide a way to expose USB device services to the Web. With this API, hardware manufacturers will be able to build cross-platform JavaScript SDKs for their devices. But most importantly this will **make USB safer and easier to use by bringing it to the Web**.

Let's see what you could expect with the WebUSB API:

1. Buy a USB device.
2. Plug it into your computer.
3. A notification appears right away, with the right website to go to for this device.
4. Simply click on it. Website is there and ready to use!
5. Click to connect and a USB device chooser shows up in Chrome, where you can pick your device.
6. Tada!

What would this procedure be like without the WebUSB API?

- Read a box, label, or search on line and possibly end up on the wrong website.
- Have to install a native application.
- Is it supported on my operating system? Make sure you download the "right" thing.
- Scary OS prompts popup and warn you about installing drivers/applications from the Internet.
- Malfunctioning code harms the whole computer. The Web is built to contain malfunctioning websites.

- Only use the USB device once? On the Web, the website is gone once you closed tab. On a computer the code sticks around.

Before we start

This article assumes you have some basic knowledge of how USB works. If not, I recommend reading [USB in a NutShell](#). For background information about USB, check out the [official USB specifications](#).

The [WebUSB API](#) [↗](#) is available in Chrome 61.

Available for Origin Trials

In order to get as much feedback as possible from developers using the WebUSB API in the field, we've previously added this feature in Chrome 54 and Chrome 57 as an [origin trial](#).

The latest trial has successfully ended in September 2017.

Privacy and security

HTTPS only

Because this API is a powerful new feature added to the Web, Chrome aims to make it available only to [secure contexts](#). This means you'll need to build with TLS in mind.

Note: We care deeply about security, so you will notice that new Web capabilities require HTTPS. The WebUSB API is no different, and is yet another good reason to get HTTPS up and running on your site.

During development you'll be able to interact with WebUSB through `http://localhost` by using tools like the [Chrome Dev Editor](#) or the handy `python -m SimpleHTTPServer`, but to deploy it on a site you'll need to have HTTPS set up on your server. I personally enjoy [GitHub Pages](#) for demo purposes.

To add HTTPS to your server you'll need to get a TLS certificate and set it up. Be sure to check out the [Security with HTTPS article](#) for best practices there. For info, you can now get free TLS certificates with the new Certificate Authority [Let's Encrypt](#) [↗](#).

User gesture required

As a security feature, getting access to connected USB devices with `navigator.usb.requestDevice` must be called via a user gesture like a touch or mouse click.


Caution: User gesture do not propagate through async events like Promises. See crbug.com/404161

Feature Policy


A feature policy is a mechanism that allows developers to selectively enable and disable various browser features and APIs. It can be defined via a HTTP header and/or an iframe "allow" attribute.

You can define a feature that controls whether the usb attribute is exposed on the Navigator object, or in other words if you allow WebUSB.

Below is an example of a header policy where WebUSB is not allowed:

Feature-Policy: fullscreen "*"; usb "none"; payment "self" https://payment. 

Below is another example of a different container policy where USB is allowed:

<iframe allowpaymentrequest allow='usb fullscreen'></iframe> 

Let's start coding

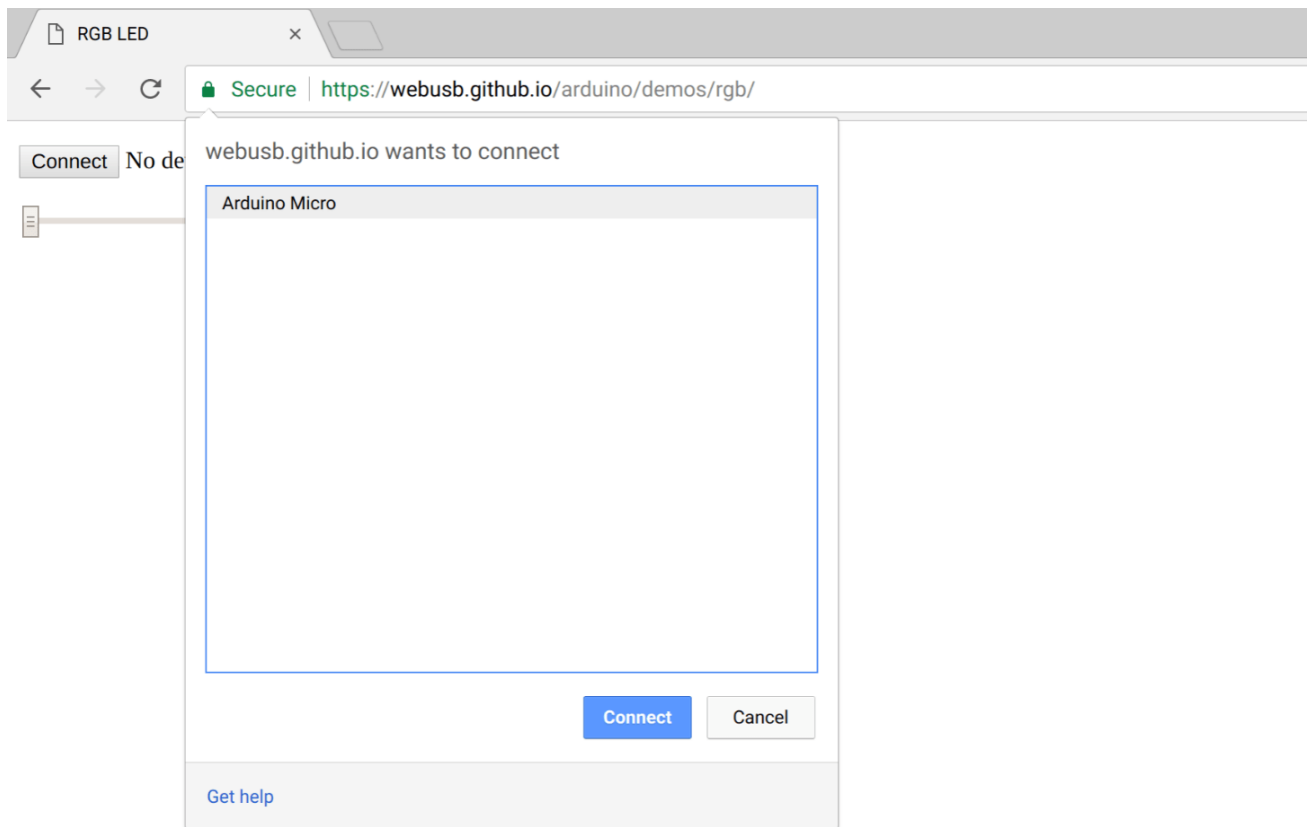
The WebUSB API relies heavily on JavaScript Promises. If you're not familiar with them, check out this great Promises tutorial. One more thing, `() => {}` are simply ECMAScript 2015 Arrow functions – they have a shorter syntax compared to function expressions and lexically bind the value of `this`.

Get access to USB devices

You can either prompt the user to select a single connected USB device using `navigator.usb.requestDevice` or call `navigator.usb.getDevices` to get a list of all connected USB devices the origin has access to.

The `navigator.usb.requestDevice` function takes a mandatory JavaScript object that defines **filters**. These filters are used to match any USB device with the given vendor

(vendorId) and optionally product (productId) identifiers. The classCode, protocolCode, serialNumber, and subclassCode keys can also be defined there as well.



For instance, here's how to get access to a connected Arduino device configured to allow the origin.

```
navigator.usb.requestDevice({ filters: [{ vendorId: 0x2341 }] })
  .then(device => {
    console.log(device.productName);      // "Arduino Micro"
    console.log(device.manufacturerName); // "Arduino LLC"
  })
  .catch(error => { console.log(error); });
```

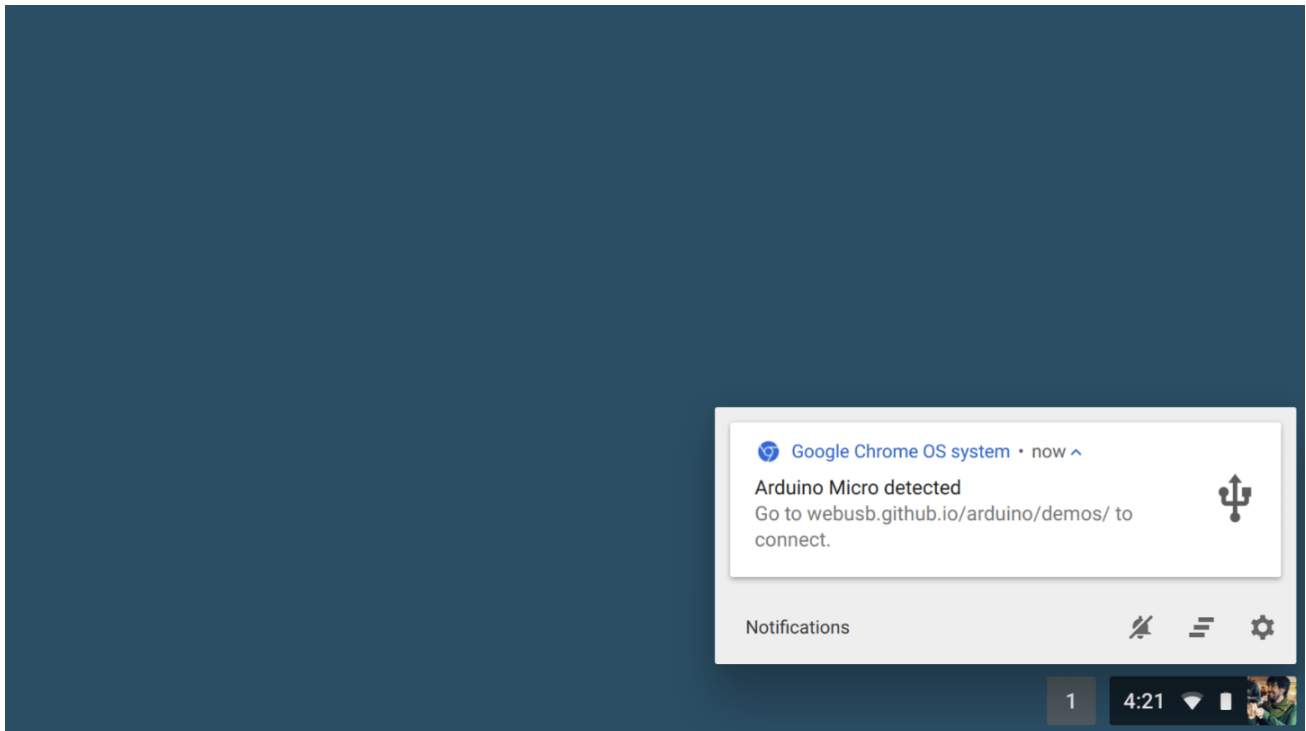


Before you ask, I didn't magically come up with this 0x2341 hexadecimal number. I simply searched for the word "Arduino" in this [List of USB ID's](#).

The USB device returned in the fulfilled promise above has some basic, yet important information about the device such as the supported USB version, maximum packet size, vendor and product IDs, the number of possible configurations the device can have - basically all fields contained in the [device USB Descriptor](#)

For info, if a USB device announces its [support for WebUSB](#), as well as defining a landing page URL, Chrome will show a persistent notification when the USB device is plugged in.

Clicking on this notification will open the landing page.



From there, you can simply call `navigator.usb.getDevices` and get access to your Arduino device as shown below.

```
navigator.usb.getDevices().then(devices => {  
  devices.map(device => {  
    console.log(device.productName);    // "Arduino Micro"  
    console.log(device.manufacturerName); // "Arduino LLC"  
  });  
})
```



Talk to an Arduino USB board

Okay, now let's see how easy it is to communicate from a WebUSB compatible Arduino board over the USB port. Check out instructions at <https://github.com/webusb/arduino> to WebUSB-enable your sketches.

Don't worry, I'll cover all the WebUSB device methods mentioned below later in this article.

```
var device;  
  
navigator.usb.requestDevice({ filters: [{ vendorId: 0x2341 }] })  
.then(selectedDevice => {  
  device = selectedDevice;  
  return device.open(); // Begin a session.
```



```

    })
    .then(() => device.selectConfiguration(1)) // Select configuration #1 for the dev
    .then(() => device.claimInterface(2)) // Request exclusive control over interface
    .then(() => device.controlTransferOut({
        requestType: 'class',
        recipient: 'interface',
        request: 0x22,
        value: 0x01,
        index: 0x02})) // Ready to receive data
    .then(() => device.transferIn(5, 64)) // Waiting for 64 bytes of data from endpoi
    .then(result => {
        let decoder = new TextDecoder();
        console.log('Received: ' + decoder.decode(result.data));
    })
    .catch(error => { console.log(error); });

```

Please keep in mind that the WebUSB library we are using here is just implementing one example protocol (based on the standard USB serial protocol) and that manufacturers can create any set and types of endpoints they wish. Control transfers are especially nice for small configuration commands as they get bus priority and have a well defined structure.

And here's the sketch that has been uploaded to the Arduino board.

```

// Third-party WebUSB Arduino library
#include <WebUSB.h>

WebUSB WebUSBSerial(1 /* https:// */, "webusb.github.io/arduino/demos");

#define Serial WebUSBSerial

void setup() {
    Serial.begin(9600);
    while (!Serial) {
        ; // Wait for serial port to connect.
    }
    Serial.write("WebUSB FTW!");
    Serial.flush();
}

void loop() {
    // Nothing here for now.
}

```

The third-party [WebUSB Arduino library](#) used in the sample code above does basically two things:

- The device acts as a WebUSB device enabling Chrome to read the [landing page URL](#).

- It exposes a WebUSB Serial API that you may use to override the default one.

Let's look at the JavaScript code again. Once we get the device picked by the user, `device.open` simply runs all platform-specific steps to start a session with the USB device. Then, we have to select an available USB Configuration with `device.selectConfiguration`. Remember that a Configuration specifies how the device is powered, its maximum power consumption and its number of interfaces. Talking about interfaces, we also need to request exclusive access with `device.claimInterface` since data can only be transferred to an interface or associated endpoints when the interface is claimed. Finally calling `device.controlTransferOut` is needed to set up the Arduino device with the appropriate commands to communicate through the WebUSB Serial API.

From there, `device.transferIn` performs a bulk transfer onto the device to inform it that the host is ready to receive bulk data. Then, the promise is fulfilled with a `result` object containing a DataView data that has to be parsed appropriately.

For those who are familiar with USB, all of this should look pretty familiar.

I want more

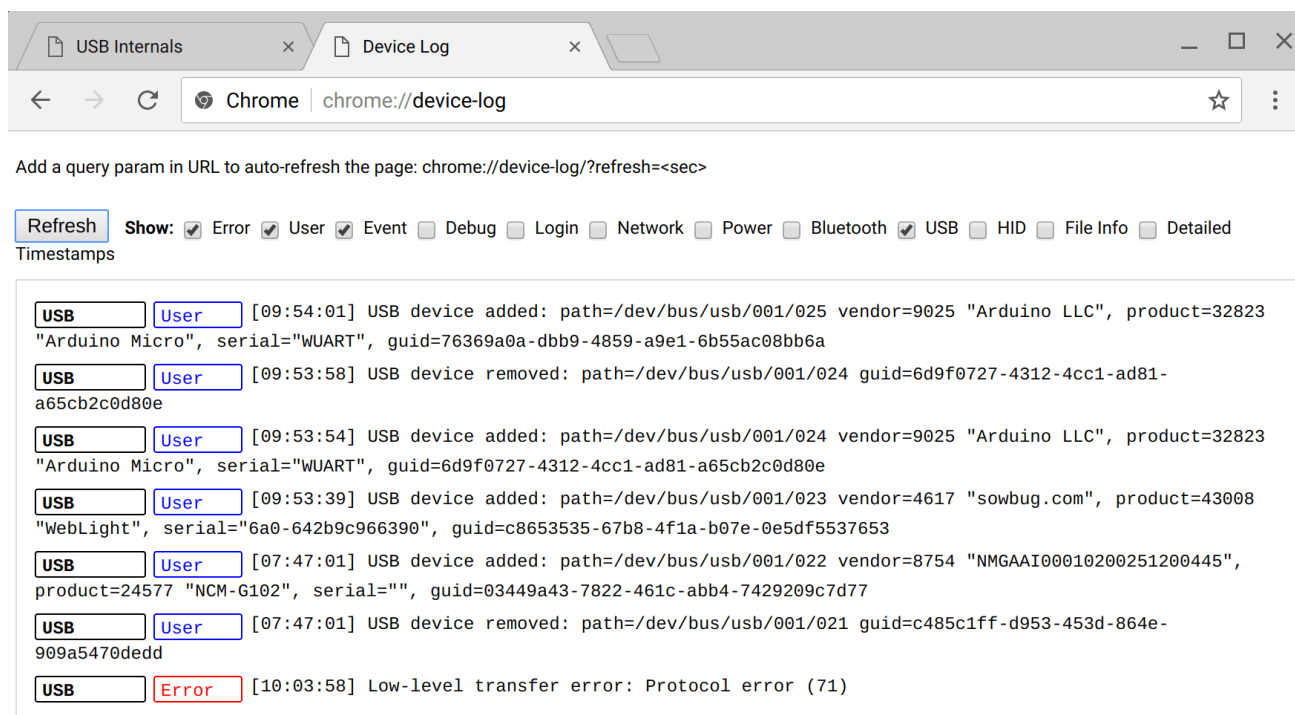
The WebUSB API lets you interact with the all USB transfer/endpoint types:

- CONTROL transfers, used to send or receive configuration or command parameters to a USB device are handled with `controlTransferIn(setup, length)` and `controlTransferOut(setup, data)`.
- INTERRUPT transfers, used for a small amount of time sensitive data are handled with the same methods as BULK transfers with `transferIn(endpointNumber, length)` and `transferOut(endpointNumber, data)`.
- ISOCHRONOUS transfers, used for streams of data like video and sound are handled with `isochronousTransferIn(endpointNumber, packetLengths)` and `isochronousTransferOut(endpointNumber, data, packetLengths)`.
- BULK transfers, used to transfer a large amount of non-time-sensitive data in a reliable way are handled with `transferIn(endpointNumber, length)` and `transferOut(endpointNumber, data)`.

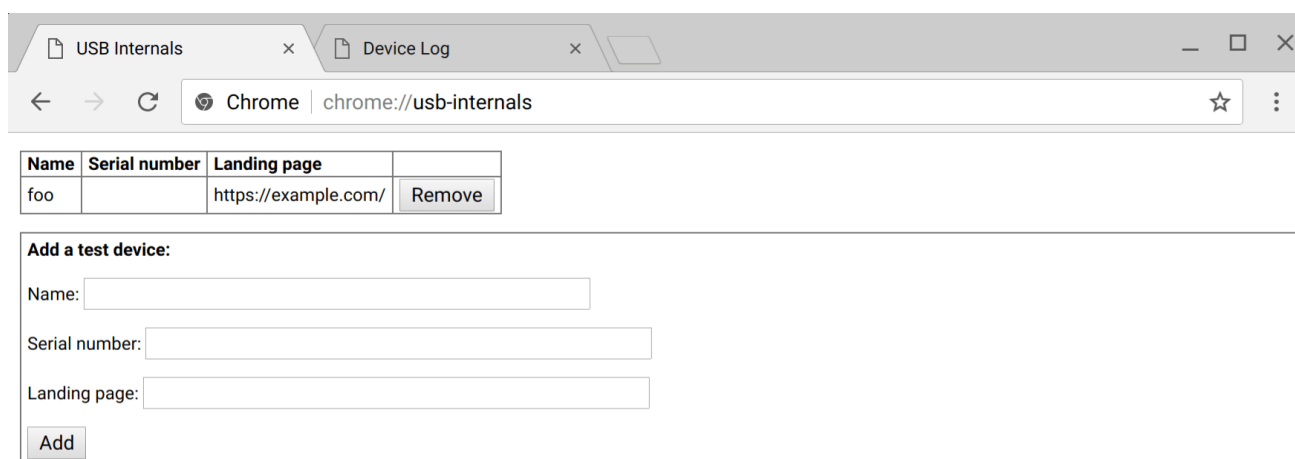
You may also want to have a look at Mike Tsao's [WebLight project](#) which provides a ground-up example of building a USB-controlled LED device designed for the WebUSB API (not using an Arduino here). You'll find hardware, software, and firmware.

Tips

Debugging USB in Chrome is easier with the internal page `chrome://device-log` where you can see all USB device related events in one single place.



The internal page `chrome://usb-internals` also comes in handy and allows you to simulate connection connection and disconnection of virtual WebUSB devices. This is be useful for doing UI testing without the need for real hardware.



On most Linux systems, USB devices are mapped with read-only permissions by default. To allow Chrome to open a USB device, you will need to add a new udev rule. Create a file at `/etc/udev/rules.d/50-yourdevicename.rules` with the following content:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="[yourdevicevendor]", MODE="0664", GROUP=
```


where [yourdevicevendor] is 2341 if your device is an Arduino for instance. ATTR{idProduct} can also be added for a more specific rule. Make sure your user is a member of the plugdev group. Then, just reconnect your device.

What's next

A second iteration of the WebUSB API will look at Shared Worker and Service Worker support. Imagine for instance a security key website using the WebUSB API that would install a service worker to act as a middle man to authenticate users.

Note: Microsoft OS 2.0 Descriptors used by the Arduino examples only work on Windows 8.1 and later. Without that Windows support still requires manual installation of an INF file.

Resources

- Stack Overflow: <https://stackoverflow.com/questions/tagged/webusb>
- WebUSB API Spec: <http://wicg.github.io/webusb/> [↗](#)
- Chrome Feature Status: <https://www.chromestatus.com/features/5651917954875392>
- Spec Issues: <https://github.com/WICG/webusb/issues>
- Implementation Bugs: <http://crbug.com?q=component:Blink>USB>
- WebUSB ♥ Arduino: <https://github.com/webusb/arduino>
- IRC: [#webusb](#) on W3C's IRC
- WICG Mailing list: <https://lists.w3.org/Archives/Public/public-wicg/>
- WebLight project: <https://github.com/sowbug/weblight>

Please share your WebUSB demos with the [#webusb](#) hashtag.

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.