Read from and write to a serial port

The Web Serial API allows websites to communicate with serial devices.

François Beaufort

(https://github.com/beaufortfrancois)

Success: The Web Serial API, part of the capabilities project (/docs/capabilities/status), launched in Chrome 89.

What is the Web Serial API?

A serial port is a bidirectional communication interface that allows sending and receiving data byte by byte.

The Web Serial API provides a way for websites to read from and write to a serial device with JavaScript. Serial devices are connected either through a serial port on the user's system or through removable USB and Bluetooth devices that emulate a serial port.

In other words, the Web Serial API bridges the web and the physical world by allowing websites to communicate with serial devices, such as microcontrollers and 3D printers.

This API is also a great companion to WebUSB (/articles/usb) as operating systems require applications to communicate with some serial ports using their higher-level serial API rather than the low-level USB API.

Suggested use cases

In the educational, hobbyist, and industrial sectors, users connect peripheral devices to their computers. These devices are often controlled by microcontrollers via a serial connection used by custom software. Some custom software to control these devices is built with web technology:

- Arduino Create (https://create.arduino.cc/)
- Betaflight Configurator (https://github.com/betaflight/betaflight-configurator)
- Espruino Web IDE (http://espruino.com/ide)
- Microsoft MakeCode (https://www.microsoft.com/en-us/makecode)

In some cases, websites communicate with the device through an agent application that users installed manually. In others, the application is delivered in a packaged application through a framework such as Electron. And in others, the user is required to perform an additional step such as copying a compiled application to the device via a USB flash drive.

In all these cases, the user experience will be improved by providing direct communication between the website and the device that it is controlling.

Current status

Step	Status
1. Create explainer	Complete (https://github.com/WICG/serial/blob/main/EXPLAINER.md)
2. Create initial draft of specification	Complete (https://github.com/WICG/serial)
3. Gather feedback & iterate on design	Complete (#feedback)
4. Origin trial	Complete (https://developers.chrome.com/origintrials/#/view_trial/2992641952387694593)
5. Launch	**Complete**

Using the Web Serial API

Feature detection

To check if the Web Serial API is supported, use:

```
if ("serial" in navigator) {
  // The Web Serial API is supported.
}
```

Open a serial port

The Web Serial API is asynchronous by design. This prevents the website UI from blocking when awaiting input, which is important because serial data can be received at any time, requiring a way to listen to it.

To open a serial port, first access a **`SerialPort`** object. For this, you can either prompt the user to select a single serial port by calling **`navigator.serial.requestPort()`** in response to a user gesture such as touch or mouse click, or pick one from **`navigator.serial.getPorts()`** which returns a list of serial ports the website has been granted access to.

```
document.querySelector('button').addEventListener('click', async () => {
  // Prompt user to select any serial port.
  const port = await navigator.serial.requestPort();
});
```

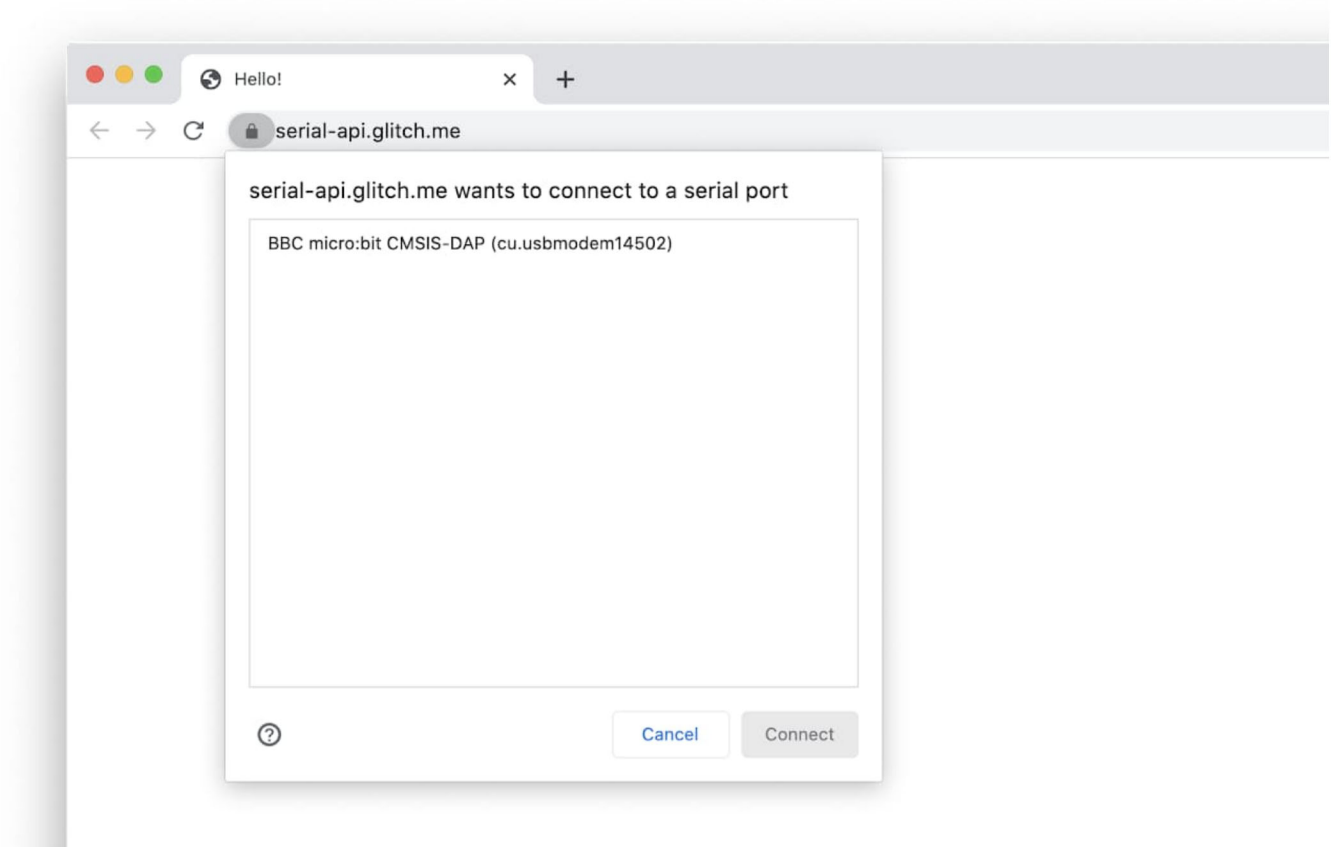
```
// Get all serial ports the user has previously granted the website access to.
const ports = await navigator.serial.getPorts();
```

The **`navigator.serial.requestPort()`** function takes an optional object literal that defines filters. Those are used to match any serial device connected over USB with a mandatory USB vendor (**`usbVendorId`**) and optional USB product identifiers (**`usbProductId`**).

```
// Filter on devices with the Arduino Uno USB Vendor/Product IDs.
const filters = [
  { usbVendorId: 0x2341, usbProductId: 0x0043 },
  { usbVendorId: 0x2341, usbProductId: 0x0001 }
];

// Prompt user to select an Arduino Uno device.
const port = await navigator.serial.requestPort({ filters });

const { usbProductId, usbVendorId } = port.getInfo();
```



User prompt for selecting a BBC micro:bit

Calling `requestPort()` prompts the user to select a device and returns a `SerialPort` object. Once you have a `SerialPort` object, calling `port.open()` with the desired baud rate will open the serial port. The `baudRate` dictionary member specifies how fast data is sent over a serial line. It is expressed in units of bits-per-second (bps). Check your device's documentation for the correct value as all the data you send and receive will be gibberish if this is specified incorrectly. For some USB and Bluetooth devices that emulate a serial port this value may be safely set to any value as it is ignored by the emulation.

```
// Prompt user to select any serial port.
const port = await navigator.serial.requestPort();

// Wait for the serial port to open.
await port.open({ baudRate: 9600 });
```

You can also specify any of the options below when opening a serial port. These options are optional and have convenient default values (https://wicg.github.io/serial/#serialoptions-dictionary).

- `dataBits`: The number of data bits per frame (either 7 or 8).
- `stopBits`: The number of stop bits at the end of a frame (either 1 or 2).
- `parity`: The parity mode (either `"none"`, `"even"` or `"odd"`).
- `bufferSize`: The size of the read and write buffers that should be created (must be less than 16MB).
- `flowControl`: The flow control mode (either `"none"` or `"hardware"`).

Read from a serial port

Input and output streams in the Web Serial API are handled by the Streams API.

Note: If streams are new to you, check out Streams API concepts (https://developer.mozilla.org/docs/Web/API/Streams_API/Concepts). This article barely scratches the surface of streams and stream handling.

After the serial port connection is established, the **`readable`** and **`writable`** properties from the **`SerialPort`** object return a ReadableStream (https://developer.mozilla.org/docs/Web/API/ReadableStream) and a WritableStream (https://developer.mozilla.org/docs/Web/API/WritableStream). Those will be used to receive data from and send data to the serial device. Both use **`Uint8Array`** instances for data transfer.

When new data arrives from the serial device, **`port.readable.getReader().read()`** returns two properties asynchronously: the **`value`** and a **`done`** boolean. If **`done`** is true, the serial port has been closed or there is no more data coming in. Calling **`port.readable.getReader()`** creates a reader and locks **`readable`** to it. While **`readable`** is locked (https://streams.spec.whatwg.org/#lock), the serial port can't be closed.

```
const reader = port.readable.getReader();

// Listen to data coming from the serial device.
while (true) {
  const { value, done } = await reader.read();
  if (done) {
    // Allow the serial port to be closed later.
    reader.releaseLock();
    break;
  }
  // value is a Uint8Array.
  console.log(value);
}
```

Some non-fatal serial port read errors can happen under some conditions such as buffer overflow, framing errors, or parity errors. Those are thrown as exceptions and can be caught by adding another loop on top of the previous one that checks **`port.readable`**. This works

because as long as the errors are non-fatal, a new ReadableStream (https://developer.mozilla.org/docs/Web/API/ReadableStream) is created automatically. If a fatal error occurs, such as the serial device being removed, then **`port.readable`** becomes null.

```
while (port.readable) {
  const reader = port.readable.getReader();

  try {
    while (true) {
      const { value, done } = await reader.read();
      if (done) {
        // Allow the serial port to be closed later.
        reader.releaseLock();
        break;
      }
      if (value) {
        console.log(value);
      }
    }
  } catch (error) {
    // TODO: Handle non-fatal read error.
  }
}
```

If the serial device sends text back, you can pipe **`port.readable`** through a **`TextDecoderStream`** as shown below. A **`TextDecoderStream`** is a transform stream (https://developer.mozilla.org/docs/Web/API/TransformStream) that grabs all **`Uint8Array`** chunks and converts them to strings.

```
const textDecoder = new TextDecoderStream();
const readableStreamClosed = port.readable.pipeTo(textDecoder.writable);
const reader = textDecoder.readable.getReader();

// Listen to data coming from the serial device.
while (true) {
  const { value, done } = await reader.read();
```

```
  if (done) {
    // Allow the serial port to be closed later.
    reader.releaseLock();
    break;
  }
  // value is a string.
  console.log(value);
}
```

You can take control of how memory is allocated when you read from the stream using a "Bring Your Own Buffer" reader. Call **`port.readable.getReader({ mode: "byob" })`** to get the ReadableStreamBYOBReader (https://developer.mozilla.org/docs/Web/API/ReadableStreamBYOBReader) interface and provide your own **`ArrayBuffer`** when calling **`read()`**. Note that the Web Serial API supports this feature in Chrome 106 or later.

```
try {
  const reader = port.readable.getReader({ mode: "byob" });
  // Call reader.read() to read data into a buffer...
} catch (error) {
  if (error instanceof TypeError) {
    // BYOB readers are not supported.
    // Fallback to port.readable.getReader()...
  }
}
```

Here's an example of how to reuse the buffer out of **`value.buffer`**:

```
const bufferSize = 1024; // 1kB
let buffer = new ArrayBuffer(bufferSize);

// Set `bufferSize` on open() to at least the size of the buffer.
await port.open({ baudRate: 9600, bufferSize });
```

```
const reader = port.readable.getReader({ mode: "byob" });
while (true) {
  const { value, done } = await reader.read(new Uint8Array(buffer));
  if (done) {
    break;
  }
  buffer = value.buffer;
  // Handle `value`.
}
```

Here's another example of how to read a specific amount of data from a serial port:

```
async function readInto(reader, buffer) {
  let offset = 0;
  while (offset < buffer.byteLength) {
    const { value, done } = await reader.read(
      new Uint8Array(buffer, offset)
    );
    if (done) {
      break;
    }
    buffer = value.buffer;
    offset += value.byteLength;
  }
  return buffer;
}

const reader = port.readable.getReader({ mode: "byob" });
let buffer = new ArrayBuffer(512);
// Read the first 512 bytes.
buffer = await readInto(reader, buffer);
// Then read the next 512 bytes.
buffer = await readInto(reader, buffer);
```

Write to a serial port

To send data to a serial device, pass data to **`port.writable.getWriter().write()`**. Calling **`releaseLock()`** on **`port.writable.getWriter()`** is required for the serial port to be closed

later.

```
const writer = port.writable.getWriter();

const data = new Uint8Array([104, 101, 108, 108, 111]); // hello
await writer.write(data);


// Allow the serial port to be closed later.
writer.releaseLock();
```

Send text to the device through a **`TextEncoderStream`** piped to **`port.writable`** as shown below.

```
const textEncoder = new TextEncoderStream();
const writableStreamClosed = textEncoder.readable.pipeTo(port.writable);

const writer = textEncoder.writable.getWriter();

await writer.write("hello");
```

Close a serial port

`port.close()` closes the serial port if its **`readable`** and **`writable`** members are unlocked (https://streams.spec.whatwg.org/#lock), meaning **`releaseLock()`** has been called for their respective reader and writer.

```
await port.close();
```

However, when continuously reading data from a serial device using a loop, **`port.readable`**

will always be locked until it encounters an error. In this case, calling **`reader.cancel()`** will force **`reader.read()`** to resolve immediately with **`{ value: undefined, done: true }`** and therefore allowing the loop to call **`reader.releaseLock()`**.

```
// Without transform streams.

let keepReading = true;
let reader;

async function readUntilClosed() {
  while (port.readable && keepReading) {
    reader = port.readable.getReader();
    try {
      while (true) {
        const { value, done } = await reader.read();
        if (done) {
          // reader.cancel() has been called.
          break;
        }
        // value is a Uint8Array.
        console.log(value);
      }
    } catch (error) {
      // Handle error...
    } finally {
      // Allow the serial port to be closed later.
      reader.releaseLock();
    }
  }

  await port.close();
}

const closedPromise = readUntilClosed();

document.querySelector('button').addEventListener('click', async () => {
  // User clicked a button to close the serial port.
  keepReading = false;
  // Force reader.read() to resolve immediately and subsequently
  // call reader.releaseLock() in the loop example above.
  reader.cancel();
  await closedPromise;
```

```
});
```

Closing a serial port is more complicated when using transform streams (/docs/capabilities/serial/like%0A%60TextDecoderStream%60%20and%20%60TextEncoderStream%60). Call `reader.cancel()` as before. Then call `writer.close()` and `port.close()`. This propagates errors through the transform streams to the underlying serial port. Because error propagation doesn't happen immediately, you need to use the `readableStreamClosed` and `writableStreamClosed` promises created earlier to detect when `port.readable` and `port.writable` have been unlocked. Cancelling the `reader` causes the stream to be aborted; this is why you must catch and ignore the resulting error.

```
// With transform streams.

const textDecoder = new TextDecoderStream();
const readableStreamClosed = port.readable.pipeTo(textDecoder.writable);
const reader = textDecoder.readable.getReader();

// Listen to data coming from the serial device.
while (true) {
  const { value, done } = await reader.read();
  if (done) {
    reader.releaseLock();
    break;
  }
  // value is a string.
  console.log(value);
}

const textEncoder = new TextEncoderStream();
const writableStreamClosed = textEncoder.readable.pipeTo(port.writable);

reader.cancel();
await readableStreamClosed.catch(() => { /* Ignore the error */ });

writer.close();
await writableStreamClosed;

await port.close();
```

Listen to connection and disconnection

If a serial port is provided by a USB device then that device may be connected or disconnected from the system. When the website has been granted permission to access a serial port, it should monitor the **`connect`** and **`disconnect`** events.

```
navigator.serial.addEventListener("connect", (event) => {
  // TODO: Automatically open event.target or warn user a port is available.
});
```

```
navigator.serial.addEventListener("disconnect", (event) => {
  // TODO: Remove |event.target| from the UI.
  // If the serial port was opened, a stream error would be observed as well.
});
```

Note: Prior to Chrome 89 the **`connect`** and **`disconnect`** events fired a custom **`SerialConnectionEvent`** object with the affected **`SerialPort`** interface available as the **`port`** attribute. You may want to use **`event.port || event.target`** to handle the transition.

Handle signals

After establishing the serial port connection, you can explicitly query and set signals exposed by the serial port for device detection and flow control. These signals are defined as boolean values. For example, some devices such as Arduino will enter a programming mode if the Data Terminal Ready (DTR) signal is toggled.

Setting output signals (https://wicg.github.io/serial/#serialoutputsignals-dictionary) and getting input signals (https://wicg.github.io/serial/#serialinputsignals-dictionary) are respectively done by calling **`port.setSignals()`** and **`port.getSignals()`**. See usage examples below.

```
// Turn off Serial Break signal.
await port.setSignals({ break: false });

// Turn on Data Terminal Ready (DTR) signal.
await port.setSignals({ dataTerminalReady: true });

// Turn off Request To Send (RTS) signal.
await port.setSignals({ requestToSend: false });
```

```
const signals = await port.getSignals();
console.log(`Clear To Send:       ${signals.clearToSend}`);
console.log(`Data Carrier Detect: ${signals.dataCarrierDetect}`);
```

```
console.log(`Data Set Ready:      ${signals.dataSetReady}`);
console.log(`Ring Indicator:      ${signals.ringIndicator}`);
```

Transforming streams

When you receive data from the serial device, you won't necessarily get all of the data at once. It may be arbitrarily chunked. For more information, see Streams API concepts (https://developer.mozilla.org/docs/Web/API/Streams_API/Concepts).

To deal with this, you can use some built-in transform streams such as **`TextDecoderStream`** or create your own transform stream which allows you to parse the incoming stream and return parsed data. The transform stream sits between the serial device and the read loop that is consuming the stream. It can apply an arbitrary transform before the data is consumed. Think of it like an assembly line: as a widget comes down the line, each step in the line modifies the widget, so that by the time it gets to its final destination, it's a fully functioning widget.

World War II Castle Bromwich Aeroplane Factory

For example, consider how to create a transform stream class that consumes a stream and chunks it based on line breaks. Its **`transform()`** method is called every time new data is received by the stream. It can either enqueue the data or save it for later. The **`flush()`** method is called when the stream is closed, and it handles any data that hasn't been processed yet.

To use the transform stream class, you need to pipe an incoming stream through it. In the third code example under Read from a serial port (#read-port), the original input stream was only piped through a **`TextDecoderStream`**, so we need to call **`pipeThrough()`** to pipe it through our new **`LineBreakTransformer`**.

```
class LineBreakTransformer {
  constructor() {
    // A container for holding stream data until a new line.
    this.chunks = "";
  }

  transform(chunk, controller) {
    // Append new chunks to existing chunks.
    this.chunks += chunk;
    // For each line breaks in chunks, send the parsed lines out.
    const lines = this.chunks.split("\r\n");
    this.chunks = lines.pop();
    lines.forEach((line) => controller.enqueue(line));
  }

  flush(controller) {
    // When the stream is closed, flush any remaining chunks out.
    controller.enqueue(this.chunks);
  }
}
```

```
const textDecoder = new TextDecoderStream();
const readableStreamClosed = port.readable.pipeTo(textDecoder.writable);
const reader = textDecoder.readable
  .pipeThrough(new TransformStream(new LineBreakTransformer()))
```

```
  .getReader();
```

For debugging serial device communication issues, use the `tee()` method of `port.readable` to split the streams going to or from the serial device. The two streams created can be consumed independently and this allows you to print one to the console for inspection.

```
const [appReadable, devReadable] = port.readable.tee();

// You may want to update UI with incoming data from appReadable
// and log incoming data in JS console for inspection from devReadable.
```

Revoke access to a serial port

The website can clean up permissions to access a serial port it is no longer interested in retaining by calling `forget()` on the `SerialPort` instance. For example, for an educational web application used on a shared computer with many devices, a large number of accumulated user-generated permissions creates a poor user experience.

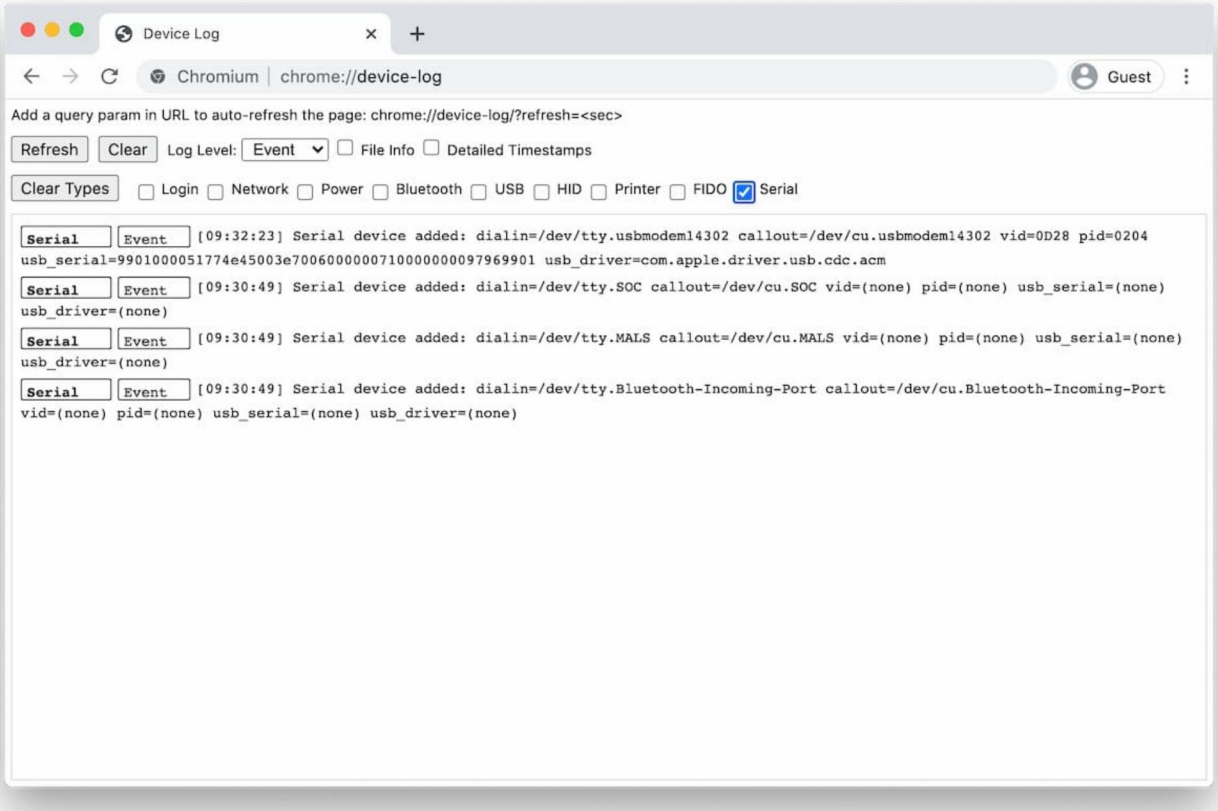
```
// Voluntarily revoke access to this serial port.
await port.forget();
```

As `forget()` is available in Chrome 103 or later, check if this feature is supported with the following:

```
if ("serial" in navigator && "forget" in SerialPort.prototype) {
  // forget() is supported.
}
```

Dev Tips

Debugging the Web Serial API in Chrome is easy with the internal page, `about://device-log` where you can see all serial device related events in one single place.



Internal page in Chrome for debugging the Web Serial API.

Codelab

In the Google Developer codelab (https://codelabs.developers.google.com/codelabs/web-serial), you'll use the Web Serial API to interact with a BBC micro:bit (https://microbit.org/) board to show images on its 5x5 LED matrix.

Browser support

The Web Serial API is available on all desktop platforms (ChromeOS, Linux, macOS, and Windows) in Chrome 89.

Polyfill

On Android, support for USB-based serial ports is possible using the WebUSB API and the Serial API polyfill (https://github.com/google/web-serial-polyfill). This polyfill is limited to hardware and platforms where the device is accessible via the WebUSB API because it has not been claimed by a built-in device driver.

Security and privacy

The spec authors have designed and implemented the Web Serial API using the core principles defined in Controlling Access to Powerful Web Platform Features (https://chromium.googlesource.com/chromium/src/+/lkgr/docs/security/permissions-for-powerful-web-platform-features.md), including user control, transparency, and ergonomics. The ability to use this API is primarily gated by a permission model that grants access to only a single serial device at a time. In response to a user prompt, the user must take active steps to select a particular serial device.

To understand the security tradeoffs, check out the security (https://wicg.github.io/serial/#security) and privacy (https://wicg.github.io/serial/#privacy) sections of the Web Serial API Explainer.

Feedback

The Chrome team would love to hear about your thoughts and experiences with the Web Serial API.

Tell us about the API design

Is there something about the API that doesn't work as expected? Or are there missing methods or properties that you need to implement your idea?

File a spec issue on the Web Serial API GitHub repo (https://github.com/wicg/serial/issues) or add your thoughts to an existing issue.

Report a problem with the implementation

Did you find a bug with Chrome's implementation? Or is the implementation different from the spec?

File a bug at https://new.crbug.com (https://issues.chromium.org/issues/new?noWizard=true&template=1964054&component=1456283). Be sure to include as much detail as you can, provide simple instructions for reproducing the bug, and have *Components* set to `Blink>Serial`. Glitch (https://glitch.com) works great for sharing quick and easy repros.

Show support

Are you planning to use the Web Serial API? Your public support helps the Chrome team prioritize features and shows other browser vendors how critical it is to support them.

Send a tweet to @ChromiumDev (https://twitter.com/chromiumdev) using the hashtag `#SerialAPI` (https://twitter.com/search?q=%23SerialAPI&src=typed_query&f=live) and let us know where and how you're using it.

Helpful links

- Specification (https://github.com/WICG/serial)
- Tracking bug (https://crbug.com/884928)
- ChromeStatus.com entry (https://chromestatus.com/feature/6577673212002304)
- Blink Component: `Blink>Serial` (https://chromestatus.com/features#component%3ABlink%3ESerial)

Demos

- Serial Terminal (https://googlechromelabs.github.io/serial-terminal/)
- WebSerial (https://webserial.io/)
- Espruino Web IDE (https://www.espruino.com/ide/)

Acknowledgements

Thanks to Reilly Grant (https://twitter.com/reillyeon) and Joe Medley (https://github.com/jpmedley) for their reviews of this article. Aeroplane factory photo by Birmingham Museums Trust (https://unsplash.com/@birminghammuseumstrust) on Unsplash (https://unsplash.com/photos/E1PSU-7aWcY).

Last updated 2020-08-12 UTC.