Web Serial API

Living document

Draft Community Group Report 29 August 2024

Latest published version:

none

Latest editor's draft:

https://wicg.github.io/serial/

Editor:

See contributors on GH

Feedback:

GitHub wicg/serial (pull requests, new issue, open issues)

Abstract

The *Serial API* provides a way for websites to read and write from a serial device through script. Such an API would bridge the web and the physical world, by allowing documents to communicate with devices such as microcontrollers, 3D printers, and other serial devices. There is also a companion explainer document.

Status of This Document

This specification was published by the Web Platform Incubator Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Contributor License Agreement (CLA) there is a limited opt-out and other conditions apply. Learn more about W3C Community and Business Groups.

This is a work in progress. All contributions welcome.

GitHub Issues are preferred for discussion of this specification.

Table of Contents

§ 1. Extensions to the `Navigator` interface

WebIDL

```
[Exposed=Window, SecureContext]
partial interface Navigator {
  [SameObject] readonly attribute Serial serial;
};
```

§ 1.1 *`serial`* attribute

When getting, the `serial` attribute always returns the same instance of the `Serial` object.

§ 2. Extensions to the `WorkerNavigator` interface

WebIDL

```
[Exposed=DedicatedWorker, SecureContext]
partial interface WorkerNavigator {
  [SameObject] readonly attribute Serial serial;
};
```

§ 2.1 *`serial`* attribute

When getting, the `serial` attribute always returns the same instance of the `Serial` object.

§ 3. `Serial` interface

WebIDL

```
[Exposed=(DedicatedWorker, Window), SecureContext]
interface Serial : EventTarget {
  attribute EventHandler onconnect;
  attribute EventHandler ondisconnect;
  Promise<sequence<SerialPort>> getPorts();
  [Exposed=Window] Promise<SerialPort> requestPort(optional
SerialPortRequestOptions options = {});
};
```

§ 3.1 *`requestPort()`* method

EXAMPLE 1

When the user first visits a site it will not have permission to access any serial devices. A site must first call `requestPort()`. This call gives the browser the opportunity to prompt the user for which device the site should be allowed to control. If the site is designed to work with a particular device which is always connected via USB the site can provide a filter restricting the devices the user can select to only those that would be compatible. For example, a site which programs Arduino-powered robots could specify a like the following to limit the set of selectable ports to only USB devices with Arduino's USB vendor ID,

```
const filter = { usbVendorId: 0x2341 };
const port = await navigator.serial.requestPort({ filters: [filter
```

If on the other hand the site expects to be used with a wide variety of devices or devices connected through a USB to serial converter it may specify no filter at all and rely on the user to select the appropriate device,

```
const port = await navigator.serial.requestPort();
```

Asking the user to choose a port requires showing a prompt to the user and so the site must have transient activation from something like the user clicking a button.

```
<button id="connect">Connect</button>

const connectButton = document.getElementById("connect");
connectButton.addEventListener('click', () => {
  try {
    const port = await navigator.serial.requestPort();
    // Continue connecting to the device attached to |port|.
  } catch (e) {
    // The prompt has been dismissed without selecting a device.
  }
});
```

The user may choose not to select a device, in which case the `Promise` will be rejected with a "`NotFoundError`" `DOMException` that the site must handle.

The `requestPort()` method steps are:

1. Let *promise* be a new promise.
2. If this's relevant global object's associated Document is not allowed to use the policy-controlled feature named "`serial`", reject *promise* with a "`SecurityError`" `DOMException` and return *promise*.
3. If the relevant global object of this does not have transient activation, reject *promise* with a "`SecurityError`" `DOMException` and return *promise*.
4. If *options*["`filters`"] is present, then for each *filter* in *options*["`filters`"] run the following steps:
    1. If *filter*["`bluetoothServiceClassId`"] is present:
        1. If *filter*["`usbVendorId`"] is present, reject *promise* with a `TypeError` and return *promise*.
        2. If *filter*["`usbProductId`"] is present, reject *promise* with a `TypeError` and return *promise*.
    2. If *filter*["`usbVendorId`"] is not present, reject *promise* with a `TypeError` and return *promise*.

    > NOTE
    > This check implements the combined rule that a `SerialPortFilter` cannot be empty and if `usbProductId` is specified then `usbVendorId` must also be specified.

5. Run the following steps in parallel:
    1. Let *allPorts* be an empty list.
    2. For each Bluetooth device registered with the system:
        1. For each `BluetoothServiceUUID` *uuid* supported by the device:
            1. If *uuid* is not a blocked Bluetooth service class UUID:
                - If *uuid* is equal to the Serial Port Profile service class ID, or
                - *options*["`allowedBluetoothServiceClassIds`"] is present and contains *uuid*:
                    1. Let *port* be a `SerialPort` representing the service on the Bluetooth device.
                    2. Append *port* to *allPorts*.
    3. For each available non-Bluetooth serial port:
        1. Let *port* be a `SerialPort` representing the port.

2. Append *port* to *allPorts*.

4. Prompt the user to grant the site access to a serial port by presenting them with a list of ports in *allPorts* that match any filter in *options*["`filters`"] if present and *allPorts* otherwise.

5. If the user does not choose a port, queue a global task on the relevant global object of this using the serial port task source to reject *promise* with a "`NotFoundError`" `DOMException` and abort these steps.

6. Let *port* be a `SerialPort` representing the port chosen by the user.

7. Queue a global task on the relevant global object of this using the serial port task source to resolve *promise* with *port*.

6. Return *promise*.

A serial port is ***available*** if it is a wired serial port and the port is physically connected to the system, or if it is a wireless serial port and the wireless device hosting the port is registered with the system.

§ 3.1.1 `SerialPortRequestOptions` dictionary

WebIDL

```
dictionary SerialPortRequestOptions {
  sequence<SerialPortFilter> filters;
  sequence<BluetoothServiceUUID>
allowedBluetoothServiceClassIds;
};
```

***`filters`* member**

Filters for serial ports

***`allowedBluetoothServiceClassIds`* member**

A list of `BluetoothServiceUUID` values representing Bluetooth service class IDs. Bluetooth ports with custom service class IDs are excluded from the list of ports presented to the user unless the service class ID is included in this list.

§ 3.1.2 `SerialPortFilter` dictionary

WebIDL

```
dictionary SerialPortFilter {
  unsigned short usbVendorId;
  unsigned short usbProductId;
  BluetoothServiceUUID bluetoothServiceClassId;
};
```

***`usbVendorId`* member**
: USB Vendor ID

***`usbProductId`* member**
: USB Product ID

***`bluetoothServiceClassId`* member**
: Bluetooth service class ID

A serial port *port* ***matches the filter*** *filter* if these steps return `true`:

1. Let *info* be the result of calling *port*.`getInfo()`.
2. If *filter*["`bluetoothServiceClassId`"] is present:
    1. If the serial port is not part of a Bluetooth device, return `false`.
    2. If *filter*["`bluetoothServiceClassId`"] is equal to *info*["`bluetoothServiceClassId`"], return `true`.
    3. Otherwise, return `false`.
3. If *filter*["`usbVendorId`"] is not present, return `true`.
4. If the serial port is not part of a USB device, return `false`.
5. If *info*["`usbVendorId`"] is not equal to *filter*["`usbVendorId`"], return `false`.
6. If *filter*["`usbProductId`"] is not present, return `true`.
7. If *info*["`usbProductId`"] is not equal to *filter*["`usbProductId`"], return `false`.
8. Otherwise, return `true`.

A serial port *port* ***matches any filter*** in a sequence of `SerialPortFilter` if these steps return `true`:

1. For each *filter* in the sequence, run these sub-steps:
    1. If *port* matches the filter *filter*, return `true`.
2. Return `false`.

§ 3.2 `getPorts()` method

EXAMPLE 2

If a serial port is provided by a USB device then that device may be connected or disconnected from the system. Once a site has permission to access a port it can receive these events and query for the set of connected devices it currently has access to.

```
// Check to see what ports are available when the page loads.
document.addEventListener('DOMContentLoaded', async () => {
  let ports = await navigator.serial.getPorts();
  // Populate the UI with options for the user to select or
  // automatically connect to devices.
});

navigator.serial.addEventListener('connect', e => {
  // Add |e.target| to the UI or automatically connect.
});

navigator.serial.addEventListener('disconnect', e => {
  // Remove |e.target| from the UI. If the device was open the
  // disconnection can also be observed as a stream error.
});
```

The `getPorts()` method steps are:

1. Let *promise* be a new promise.
2. If this's relevant global object's associated Document is not allowed to use the policy-controlled feature named `"serial"`, reject *promise* with a `"SecurityError"` `DOMException` and return *promise*.
3. Run the following steps in parallel:

1. Let *availablePorts* be the sequence of available serial ports which the user has allowed the site to access as the result of a previous call to `requestPort()`.
2. Let *ports* be the sequence of the `SerialPort`s representing the ports in *availablePorts*.
3. Queue a global task on the relevant global object of this using the serial port task source to resolve *promise* with *ports*.

4. Return *promise*.

§ 3.3 `onconnect` attribute

`onconnect` is an event handler IDL attribute for the ***connect*** event type.

§ 3.4 `ondisconnect` attribute

`ondisconnect` is an event handler IDL attribute for the ***disconnect*** event type.

§ 4. `SerialPort` interface

WebIDL

```
[Exposed=(DedicatedWorker,Window), SecureContext]
interface SerialPort : EventTarget {
  attribute EventHandler onconnect;
  attribute EventHandler ondisconnect;
  readonly attribute boolean connected;
  readonly attribute ReadableStream readable;
  readonly attribute WritableStream writable;

  SerialPortInfo getInfo();

  Promise<undefined> open(SerialOptions options);
  Promise<undefined> setSignals(optional SerialOutputSignals
signals = {});
  Promise<SerialInputSignals> getSignals();
  Promise<undefined> close();
```

```
  Promise<undefined> forget();
};
```

Methods on this interface typically complete asynchronously, queuing work on the ***serial port task source***.

The get the parent algorithm for `SerialPort` returns the same `Serial` instance that is returned by the `SerialPort`'s relevant global object's `Navigator` object's `serial` getter.

Instances of `SerialPort` are created with the internal slots described in the following table:

Internal slot	Initial value	Description (non-normative)
[[state]]	`"closed"`	Tracks the active state of the `SerialPort`
[[bufferSize]]	undefined	The amount of data to buffer for transmit and receive
[[connected]]	`false`	A flag indicating the logical connection state of serial port
[[readable]]	`null`	A `ReadableStream` that receives data from the port
[[readFatal]]	`false`	A flag indicating that the port has encountered a fatal read error
[[writable]]	`null`	A `WritableStream` that transmits data to the port
[[writeFatal]]	`false`	A flag indicating that the port has encountered a fatal write error
[[pendingClosePromise]]	`null`	A `Promise` used to wait for `readable` and `writable` to close

§ 4.1 ***onconnect*** attribute

`onconnect` is an event handler IDL attribute for the `connect` event type.

When a serial port that the user has allowed the site to access as the result of a previous call to `requestPort()` becomes logically connected, run the following steps:

1. Let *port* be a `SerialPort` representing the port.
2. Set *port*.`[[connected]]` to `true`.
3. Fire an event named `connect` at *port* with its `bubbles` attribute initialized to `true`.

A serial port is ***logically connected*** if it is a wired serial port and the port is physically connected to the system, or if it is a wireless serial port and the system has active connections to the wireless device (e.g. an open Bluetooth L2CAP channel).

§ 4.2 `ondisconnect` attribute

`ondisconnect` is an event handler IDL attribute for the `disconnect` event type.

When a serial port that the user has allowed the site to access as the result of a previous call to `requestPort()` is no longer logically connected, run the following steps:

1. Let *port* be a `SerialPort` representing the port.
2. Set *port*.`[[connected]]` to `false`.
3. Fire an event named `disconnect` at *port* with its `bubbles` attribute initialized to `true`.

§ 4.3 `getInfo()` method

The `getInfo()` method steps are:

1. Let *info* be a new `SerialPortInfo` dictionary.
2. If the port is part of a USB device, perform the following steps:
    1. Set *info*["`usbVendorId`"] to the vendor ID of the device.
    2. Set *info*["`usbProductId`"] to the product ID of the device.
3. If the port is a service on a Bluetooth device, perform the following steps:
    1. Set *info*["`bluetoothServiceClassId`"] to the service class UUID of the Bluetooth service.
4. Return *info*.

§ 4.3.1 *SerialPortInfo* dictionary

WebIDL

```
dictionary SerialPortInfo {
  unsigned short usbVendorId;
  unsigned short usbProductId;
  BluetoothServiceUUID bluetoothServiceClassId;
};
```

***usbVendorId* member**

If the port is part of a USB device this member will be the 16-bit vendor ID of that device. Otherwise it will be `undefined`.

***usbProductId* member**

If the port is part of a USB device this member will be the 16-bit product ID of that device. Otherwise it will be `undefined`.

***bluetoothServiceClassId* member**

If the port is a service on a Bluetooth device this member will be a `BluetoothServiceUUID` containing the service class UUID. Otherwise it will be `undefined`.

§ 4.4 *open()* method

EXAMPLE 3

Before communicating on a serial port it must be opened. Opening the port allows the site to specify the necessary parameters which control how data is transmitted and received. Developers should check the documentation for the device they are connecting to for the appropriate parameters.

```
await port.open({ baudRate: /* pick your baud rate */ });
```

Once `open()` has resolved the `readable` and `writable` attributes can be accessed to get the `ReadableStream` and `WritableStream` instances for receiving data from and sending data to the connected device.

The `open()` method steps are:

1. Let *promise* be a new promise.
2. If this.`[[state]]` is not `"closed"`, reject *promise* with an "`InvalidStateError`" `DOMException` and return *promise*.
3. If *options*["`dataBits`"] is not 7 or 8, reject *promise* with `TypeError` and return *promise*.
4. If *options*["`stopBits`"] is not 1 or 2, reject *promise* with `TypeError` and return *promise*.
5. If *options*["`bufferSize`"] is 0, reject *promise* with `TypeError` and return *promise*.
6. Optionally, if *options*["`bufferSize`"] is larger than the implementation is able to support, reject *promise* with a `TypeError` and return *promise*.
7. Set this.`[[state]]` to `"opening"`.
8. Perform the following steps in parallel.
    1. Invoke the operating system to open the serial port using the connection parameters (or their defaults) specified in *options*.
    2. If this fails for any reason, queue a global task on the relevant global object of this using the serial port task source to reject *promise* with a "`NetworkError`" `DOMException` and abort these steps.
    3. Set this.`[[state]]` to `"opened"`.
    4. Set this.`[[bufferSize]]` to *options*["`bufferSize`"].

5. Queue a global task on the relevant global object of this using the serial port task source to resolve *promise* with `undefined`.

9. Return *promise*.

§ 4.4.1 `SerialOptions` dictionary

WebIDL

```
dictionary SerialOptions {
  [EnforceRange] required unsigned long baudRate;
  [EnforceRange] octet dataBits = 8;
  [EnforceRange] octet stopBits = 1;
  ParityType parity = "none";
  [EnforceRange] unsigned long bufferSize = 255;
  FlowControlType flowControl = "none";
};
```

***`baudRate`* member**

A positive, non-zero value indicating the baud rate at which serial communication should be established.

> NOTE
> `baudRate` is the only required member of this dictionary. While there are common default for other connection parameters it is important for developers to consider and consult with the documentation for devices they intend to connect to determine the correct values. While some values are common there is no standard baud rate. Requiring this parameter reduces the potential for confusion if an arbitrary default were chosen by this specification.

***`dataBits`* member**

The number of data bits per frame. Either 7 or 8.

***`stopBits`* member**

The number of stop bits at the end of a frame. Either 1 or 2.

***`parity`* member**

The parity mode.

***`bufferSize`* member**

A positive, non-zero value indicating the size of the read and write buffers that should be created.

***flowControl* member**

The flow control mode.

§ *4.4.1.1 `ParityType` enum*

WebIDL

```
enum ParityType {
  "none",
  "even",
  "odd"
};
```

none

No parity bit is sent for each data word.

even

Data word plus parity bit has even parity.

odd

Data word plus parity bit has odd parity.

§ *4.4.1.2 `FlowControlType` enum*

WebIDL

```
enum FlowControlType {
  "none",
  "hardware"
};
```

none

No flow control is enabled.

hardware

Hardware flow control using the RTS and CTS signals is enabled.

§ 4.5 *connected* attribute

The connected getter steps are:

1. Return this.[[connected]].

§ 4.6 *readable* attribute

EXAMPLE 4

An application receiving data from a serial port will typically use a nested pair of loops like this,

```
while (port.readable) {
  const reader = port.readable.getReader();
  try {
    while (true) {
      const { value, done } = await reader.read();
      if (done) {
        // |reader| has been canceled.
        break;
      }
      // Do something with |value|...
    }
  } catch (error) {
    // Handle |error|...
  } finally {
    reader.releaseLock();
  }
}
```

The inner loop will read chunks of data from the port until an error is encountered, at which point the code in the "catch" block will be executed. The outer loop handles recoverable errors such as parity check failures by opening a new reader. Fatal errors will cause `readable` to become `null` and the loop to end.

As long as the serial port is open it can continue to produce data and the amount of data in each of the chunks returned by `read()` will be essentially arbitrary based on the timing of when it is called. It is up to the device and the code communicating with it to decide what constitutes a complete message. For example, a device might communicate with the host using ASCII-formatted text where each message ends with a newline (or the sequence `"\r\n"`). A pipeline of `TransformStream`s can be used to automatically convert the `Uint8Array` chunks provided by `readable` into `DOMString`s containing an entire line of text each.

```
class LineBreakTransformer {
  constructor() {
    this.container = '';
  }
```

```
  transform(chunk, controller) {
    this.container += chunk;
    const lines = this.container.split('\r\n');
    this.container = lines.pop();
    lines.forEach(line => controller.enqueue(line));
  }

  flush(controller) {
    controller.enqueue(this.container);
  }
}

const lineReader = port.readable
    .pipeThrough(new TextDecoderStream())
    .pipeThrough(new TransformStream(new LineBreakTransformer()))
    .getReader();
```

Some other ways of encoding message boundaries are to prefix each message with its length or to wait a defined length of time before transmitting the next message. Implementing a `TransformStream` for these types of message boundaries is left as an exercise for the reader.

While the `read()` method is asynchronous and does not block execution, in code using async/await syntax it can seem as if it does. In this situation it may be helpful to implement a timeout which will allow the code to continue execution if no data is received for a period of time. The example below uses the `releaseLock()` method to interrupt a call to `read()` after a timer expires. This will not close the stream and so any data received after the timeout can still be read later after calling `getReader()` again.

```
async function readWithTimeout(port, timeout) {
  const reader = port.readable.getReader();
  const timer = setTimeout(() => {
    reader.releaseLock();
  }, timeout);
  const result = await reader.read();
  clearTimeout(timer);
  reader.releaseLock();
  return result;
}
```

> This feature of `releaseLock()` was added in whatwg/streams#1168 and has only recently been implemented by browsers.

The `readable` getter steps are:

1. If this.`[[readable]]` is not `null`, return this.`[[readable]]`.
2. If this.`[[state]]` is not `"opened"`, return `null`.
3. If this.`[[readFatal]]` is `true`, return `null`.
4. Let *stream* be a new `ReadableStream`.
5. Let *pullAlgorithm* be the following steps:
    1. Let *desiredSize* be the desired size to fill up to the high water mark for this.`[[readable]]`.
    2. If this.`[[readable]]`'s current BYOB request view is non-null, then set *desiredSize* to this.`[[readable]]`'s current BYOB request view's byte length.
    3. Run the following steps in parallel:
        1. Invoke the operating system to read up to *desiredSize* bytes from the port, putting the result in the byte sequence *bytes*.

            > NOTE
            > this.`[[state]]` becoming `"forgotten"` must be treated as if *the port was disconnected*.

        2. Queue a global task on the relevant global object of this using the serial port task source to run the following steps:
            1. If no errors were encountered, then:
                1. If this.`[[readable]]`'s current BYOB request view is non-null, then write *bytes* into this.`[[readable]]`'s current BYOB request view, and set *view* to this.`[[readable]]`'s current BYOB request view.
                2. Otherwise, set *view* to the result of creating a `Uint8Array` from *bytes* in this's relevant Realm.
                3. Enqueue *view* into this.`[[readable]]`.
            2. If a buffer overrun condition was encountered, invoke error on

this.[[readable]] with a "BufferOverrunError" DOMException and invoke the steps to handle closing the readable stream.

3. If a break condition was encountered, invoke error on this.[[readable]] with a "BreakError" DOMException and invoke the steps to handle closing the readable stream.
4. If a framing error was encountered, invoke error on this.[[readable]] with a "FramingError" DOMException and invoke the steps to handle closing the readable stream.
5. If a parity error was encountered, invoke error on this.[[readable]] with a "ParityError" DOMException and invoke the steps to handle closing the readable stream.
6. If an operating system error was encountered, invoke error on this.[[readable]] with an "UnknownError" DOMException and invoke the steps to handle closing the readable stream.
7. If *the port was disconnected*, run the following steps:
    1. Set this.[[readFatal]] to true,
    2. Invoke error on this.[[readable]] with a "NetworkError" DOMException.
    3. Invoke the steps to handle closing the readable stream.

4. Return a promise resolved with undefined.

> NOTE
>
> The Promise returned by this algorithm is immediately resolved so that it does not block canceling the stream. [STREAMS] specifies that this algorithm will not be invoked again until a chunk is enqueued.

6. Let *cancelAlgorithm* be the following steps:
    1. Let *promise* be a new promise.
    2. Run the following steps in parallel.
        1. Invoke the operating system to discard the contents of all software and hardware receive buffers for the port.
        2. Queue a global task on the relevant global object of this using the serial port task source to run the following steps:

1. Invoke the steps to handle closing the readable stream.
2. Resolve *promise* with `undefined`.

3. Return *promise*.

7. Set up with byte reading support *stream* with *pullAlgorithm* set to *pullAlgorithm*, *cancelAlgorithm* set to *cancelAlgorithm*, and *highWaterMark* set to this.`[[bufferSize]]`.
8. Set this.`[[readable]]` to *stream*.
9. Return *stream*.

To ***handle closing the readable stream*** perform the following steps:

1. Set this.`[[readable]]` to `null`.
2. If this.`[[writable]]` is `null` and this.`[[pendingClosePromise]]` is not `null`, resolve this.`[[pendingClosePromise]]` with `undefined`.

§ 4.7 `writable` attribute

> EXAMPLE 5
>
> To write individual chunks of data to the port a `WritableStreamDefaultWriter` can be created and released as necessary. This example uses a `TextEncoder` to encode a `DOMString` as the necessary `Uint8Array` for transmission.
>
> ```
> const encoder = new TextEncoder();
> const writer = port.writable.getWriter();
> await writer.write(encoder.encode("PING"));
> writer.releaseLock();
> ```
>
> When writing larger chunks it can be important to allow the port to apply back pressure so that the serial transmitter does not get too far behind sending data generated by the application. The `write()` method returns a `Promise` which resolves when data has been written. While having some data available in the transmit buffer is important to maintain good throughput awaiting this `Promise` before generating too many chunks of data is a good practice to avoid excessive buffering.

The `writable` getter steps are:

1. If this.`[[writable]]` is not `null`, return this.`[[writable]]`.
2. If this.`[[state]]` is not `"opened"`, return `null`.
3. If this.`[[writeFatal]]` is `true`, return `null`.
4. Let *stream* be a new `WritableStream`.
5. Let *signal* be *stream*'s signal.
6. Let *writeAlgorithm* be the following steps, given *chunk*:
    1. Let *promise* be a new promise.
    2. Assert: *signal* is not aborted.
    3. If *chunk* cannot be converted to an IDL value of type `BufferSource`, reject *promise* with a `TypeError` and return *promise*. Otherwise, save the result of the conversion in *source*.
    4. Get a copy of the buffer source *source* and save the result in *bytes*.
    5. In parallel, run the following steps:
        1. Invoke the operating system to write *bytes* to the port. Alternately, store the chunk for future coalescing.

            > NOTE
            > The operating system may return from this operation once *bytes* has been queued for transmission rather than after it has been transmitted.

            > NOTE
            > this.`[[state]]` becoming `"forgotten"` must be treated as if *the port was disconnected*.

        2. Queue a global task on the relevant global object of this using the serial port task source to run the following steps:
            1. If the chunk was successfully written, or was stored for future coalescing, resolve *promise* with `undefined`.

> NOTE
> [STREAMS] specifies that *writeAlgorithm* will only be invoked after the `Promise` returned by a previous invocation of this algorithm has resolved. For efficiency an implementation is allowed to resolve this `Promise` early in order to coalesce multiple chunks waiting in the `WritableStream`'s internal queue into a single request to the operating system.

2. If an operating system error was encountered, reject *promise* with an "`UnknownError`" `DOMException`.
3. If *the port was disconnected*, run the following steps:
    1. Set this.`[[writeFatal]]` to `true`.
    2. Reject *promise* with a "`NetworkError`" `DOMException`.
    3. Invoke the steps to handle closing the writable stream.
4. If *signal* is aborted, reject *promise* with *signal*'s abort reason.

6. Return *promise*.

7. Let *abortAlgorithm* be the following steps:
    1. Let *promise* be a new promise.
    2. Run the following steps in parallel.
        1. Invoke the operating system to discard the contents of all software and hardware transmit buffers for the port.
        2. Queue a global task on the relevant global object of this using the serial port task source to run the following steps:
            1. Invoke the steps to handle closing the writable stream.
            2. Resolve *promise* with `undefined`.
    3. Return *promise*.
8. Let *closeAlgorithm* be the following steps:
    1. Let *promise* be a new promise.
    2. Run the following steps in parallel.
        1. Invoke the operating system to flush the contents of all software and hardware transmit buffers for the port.
        2. Queue a global task on the relevant global object of this using the

serial port task source to run the following steps:

1. Invoke the steps to handle closing the writable stream.
2. If *signal* is aborted, reject *promise* with *signal*'s abort reason.
3. Otherwise, resolve *promise* with `undefined`.

3. Return *promise*.

9. Set up *stream* with writeAlgorithm set to *writeAlgorithm*, abortAlgorithm set to *abortAlgorithm*, closeAlgorithm set to *closeAlgorithm*, highWaterMark set to this.`[[bufferSize]]`, and sizeAlgorithm set to a byte-counting size algorithm.
10. Add the following abort steps to *signal*:
    1. Cause any invocation of the operating system to write to the port to return as soon as possible no matter how much data has been written.
11. Set this.`[[writable]]` to *stream*.
12. Return *stream*.

To ***handle closing the writable stream*** perform the following steps:

1. Set this.`[[writable]]` to `null`.
2. If this.`[[readable]]` is `null` and this.`[[pendingClosePromise]]` is not `null`, resolve this.`[[pendingClosePromise]]` with `undefined`.

§ 4.8 `setSignals()` method

> EXAMPLE 6
>
> Serial ports include a number of additional signals for device detection and flow control which can be queried and set explicitly. As an example, programming some micro-controllers first requires entering a "programming" mode by toggling the "Data Terminal Ready" (or DTR) signal.
>
> ```
> await port.setSignals({ dataTerminalReady: false });
> await new Promise(resolve => setTimeout(resolve, 200));
> await port.setSignals({ dataTerminalReady: true });
> ```

The `setSignals()` method steps are:

1. Let *promise* be a new promise.
2. If this.[[state]] is not "opened", reject *promise* with an "InvalidStateError" DOMException and return *promise*.
3. If all of the specified members of *signals* are not present reject *promise* with TypeError and return *promise*.
4. Perform the following steps in parallel:
    1. If *signals*["dataTerminalReady"] is present, invoke the operating system to either assert (if `true`) or deassert (if `false`) the "data terminal ready" or "DTR" signal on the serial port.
    2. If *signals*["requestToSend"] is present, invoke the operating system to either assert (if `true`) or deassert (if `false`) the "request to send" or "RTS" signal on the serial port.
    3. If *signals*["break"] is present, invoke the operating system to either assert (if `true`) or deassert (if `false`) the "break" signal on the serial port.

        > NOTE
        > The "break" signal is typically implemented as an in-band signal by holding the transmit line at the "mark" voltage and thus prevents data transmission for as long as it remains asserted.

    4. If the operating system fails to change the state of any of these signals for any reason, queue a global task on the relevant global object of this using the serial port task source to reject *promise* with a "NetworkError" DOMException.
    5. Queue a global task on the relevant global object of this using the serial port task source to resolve *promise* with `undefined`.
5. Return *promise*.

§ 4.8.1 *SerialOutputSignals* dictionary

WebIDL

```
dictionary SerialOutputSignals {
  boolean dataTerminalReady;
  boolean requestToSend;
```

```
  boolean break;
};
```

dataTerminalReady
: Data Terminal Ready (DTR)

requestToSend
: Request To Send (RTS)

break
: Break

§ 4.9 `getSignals()` method

The `getSignals()` method steps are:

1. Let *promise* be a new promise.
2. If this.`[[state]]` is not `"opened"`, reject *promise* with an "`InvalidStateError`" `DOMException` and return *promise*.
3. Perform the following steps in parallel:
    1. Query the operating system for the status of the control signals that may be asserted by the device connected to the serial port.
    2. If the operating system fails to determine the status of these signals for any reason, queue a global task on the relevant global object of this using the serial port task source to reject *promise* with a "`NetworkError`" `DOMException` and abort these steps.
    3. Let *signals* be a new `SerialInputSignals`.
    4. Set *signals*["`dataCarrierDetect`"] to `true` if the "data carrier detect" or "DCD" signal has been asserted by the device, and `false` otherwise.
    5. Set *signals*["`clearToSend`"] to `true` if the "clear to send" or "CTS" signal has been asserted by the device, and `false` otherwise.
    6. Set *signals*["`ringIndicator`"] to `true` if the "ring indicator" or "RI" signal has been asserted by the device, and `false` otherwise.
    7. Set *signals*["`dataSetReady`"] to `true` if the "data set ready" or "DSR" signal has been asserted by the device, and `false` otherwise.
    8. Queue a global task on the relevant global object of this using the serial

port task source to resolve *promise* with *signals*.

4. Return *promise*.

§ 4.9.1 `SerialInputSignals` dictionary

WebIDL

```
dictionary SerialInputSignals {
  required boolean dataCarrierDetect;
  required boolean clearToSend;
  required boolean ringIndicator;
  required boolean dataSetReady;
};
```

***`dataCarrierDetect`* member**
: Data Carrier Detect (DCD)

***`clearToSend`* member**
: Clear To Send (CTS)

***`ringIndicator`* member**
: Ring Indicator (RI)

***`dataSetReady`* member**
: Data Set Ready (DSR)

§ 4.10 `close()` method

EXAMPLE 7

When communication with the port is no longer required it can be closed and the associated resources released by the system.

Calling `port.close()` implicitly invokes `port.readable.cancel()` and `port.writable.abort()` in order to clear any buffered data. If the application has called `port.readable.getReader()` or `port.writable.getWriter()` the stream is locked and the port cannot be closed. This forces the developer to decide how to handle any read or write operations that are in progress. For example, to ensure that all buffered data has been transmitted before the port is closed the application must await the `Promise` returned by `writer.close()`.

```
const encoder = new TextEncoder();
const writer = port.writable.getWriter();
writer.write(encoder.encode("A long message that will take..."));
await writer.close();
await port.close();
```

To discard any unsent data the application could instead call `writer.abort()`.

If a `TransformStream` is being piped to `port.writable` then waiting for the `Promise` returned by `writer.close()` to resolve is insufficient. The application must wait for the pipe chain to close by waiting for the `Promise` returned by `pipeTo()` to resolve instead.

```
const encoder = new TextEncoderStream();
const writableStreamClosed = encoder.readable.pipeTo(port.writable
const writer = encoder.writable.getWriter();
writer.write("A long message that will take...");
writer.close();
await writableStreamClosed;
await port.close();
```

If a loop is being used to read chunks from the port, as is done in Example 4, then it must be exited before calling `port.close()`.

```
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| has been canceled.
          break;
        }
        // Do something with |value|...
      }
    } catch (error) {
      // Handle |error|...
    } finally {
      reader.releaseLock();
    }
  }

  await port.close();
}

const closed = readUntilClosed();

// Sometime later...
keepReading = false;
reader.cancel();
await closed;
```

Calling `reader.cancel()` causes the call to `reader.read()` to return immediately, exiting the inner loop and calling `reader.releaseLock()`. The outer loop then exits because `keepReading` has been set to `false` and with the stream unlocked `port.close()` can complete successfully.

While it is also possible to call `port.close()` immediately after awaiting the `Promise` returned by `reader.cancel()` it is better to place the call to `port.close()` as the last step of `readUntilClosed()` so that the port is also closed when a fatal error is encountered and `port.readable` becomes `null`.

The `close()` method steps are:

1. Let *promise* be a new promise.

2. If this.[[state]] is not "opened", reject *promise* with an "InvalidStateError" DOMException and return *promise*.
3. Let *cancelPromise* be the result of invoking cancel on this.[[readable]] or a promise resolved with undefined if this.[[readable]] is null.
4. Let *abortPromise* be the result of invoking abort on this.[[writable]] or a promise resolved with undefined if this.[[writable]] is null.
5. Let *pendingClosePromise* be a new promise.
6. If this.[[readable]] and this.[[writable]] are null, resolve *pendingClosePromise* with undefined.
7. Set this.[[pendingClosePromise]] to *pendingClosePromise*.
8. Let *combinedPromise* be the result of getting a promise to wait for all with «*cancelPromise*, *abortPromise*, *pendingClosePromise*».
9. Set this.[[state]] to "closing".
10. React to *combinedPromise*.
    - If *combinedPromise* was fulfilled, then:
        1. Run the following steps in parallel:
            1. Invoke the operating system to close the serial port and release any associated resources.
            2. Set this.[[state]] to "closed".
            3. Set this.[[readFatal]] and this.[[writeFatal]] to false.
            4. Set this.[[pendingClosePromise]] to null.
            5. Queue a global task on the relevant global object of this using the serial port task source to resolve *promise* with undefined.
    - If *combinedPromise* was rejected with reason *r*, then:
        1. Set this.[[pendingClosePromise]] to null.
        2. Queue a global task on the relevant global object of this using the serial port task source to reject *promise* with *r*.
11. Return *promise*.

§ 4.11 `forget()` method

EXAMPLE 8

It is posssible to voluntarily revoke a permission to a serial port that was granted by a user.

```
// Request a serial port.
const port = await navigator.serial.requestPort();

// Then later... revoke permission to the serial port.
await port.forget();
```

The `forget()` method steps are:

1. If the user agent can't perform this action (e.g. permission was granted by administrator policy), return a promise resolved with `undefined`.
2. Run the following steps in parallel:
    1. Set this.`[[state]]` to `"forgetting"`.
    2. Remove this from the sequence of serial ports on the system which the user has allowed the site to access as the result of a previous call to `requestPort()`.
    3. Set this.`[[state]]` to `"forgotten"`.
    4. Queue a global task on the relevant global object of this using the serial port task source to resolve *promise* with `undefined`.
3. Return *promise*.

§ 5. Blocklist

This specification relies on a blocklist file in the https://github.com/WICG/serial repository to restrict the set of ports a website can access.

The result of ***parsing the Bluetooth service class ID blocklist*** at a URL *url* is a list of `UUID` values representing custom service IDs.

The ***Serial Port Profile service class ID*** is a `BluetoothServiceUUID` with value `"00001101-0000-1000-8000-00805f9b34fb"`.

A {{BluetoothServiceUUID}} *serviceUuid* is a ***blocked Bluetooth service class UUID*** if the following steps return `true`:

1. Let *uuid* be the result of calling `BluetoothUUID.getService()` with *serviceUuid*.
2. Let *blocklist* be the result of parsing the Bluetooth service class ID blocklist at https://github.com/WICG/serial/blob/main/blocklist.txt.
3. If *blocklist* contains *uuid*, return `true`.
4. If *uuid* is the Serial Port Profile service class ID, return `false`.
5. If *uuid* ends with `"-0000-1000-8000-00805f9b34fb"`, return `true`.
6. Otherwise, return `false`.

§ 6. Integrations

§ 6.1 Permissions Policy

This specification defines a feature that controls whether the methods exposed by the `serial` attribute on the `Navigator` object may be used.

The feature name for this feature is "***`serial`***"`.

The default allowlist for this feature is `'self'`.

§ 7. Security considerations

This section is non-normative.

This API poses similar a security risk to [WEB-BLUETOOTH] and [WEBUSB] and so lessons from those are applicable here. The primary threats are:

- A malicious site that has tricked the user into granting it access to a device using the device's intended capabilities for malicious purposes. For example, a robot causing physical damage.

- A malicious site that has tricked the user into granting it access to a device installing its own firmware into the device in order to modify the device's intended capabilities for malicious purposes or to attack the host to which it is connected. For example, triggering a buffer overflow in other host software which communicates with the device.
- Malicious code injected into a trusted site which has been granted access to the device doing any of the above. For example, an online firmware update utility being hacked to deliver malicious firmware.
- An attacker convincing the user to connect a malicious device to their system which colludes with a malicious or exploited site to create a web-based channel for communicating back to the attacker.

The primary mitigation to all of these attacks is the `requestPort()` pattern, which requires user interaction and only supports granting access to a single device at a time. This prevents drive-by attacks because a site cannot enumerate all connected devices to determine whether a vulnerable device exists and must instead proactively inform the user that it desires access. Implementations may also provide a visual indication that a site is currently communicating with a device and controls for revoking that permission at any time.

This specification requires the site to be served from a secure context in order to prevent malicious code from being injected by a network-based attacker. This ensures that the site identity shown to the user when making permission decisions is accurate. This specification also requires top-level documents to opt-in through [PERMISSIONS-POLICY] before allowing a cross-origin iframe to use the API. When combined with [CSP3] these mechanisms provide protection against malicious code injection attacks.

The remaining concern is the exploitation of a connected device through a phishing attack that convinces the user to grant a malicious site access to a device. These attacks can be used to either exploit the device’s capabilities as designed or to install malicious firmware on the device that will in turn attack the host computer. Host software may be vulnerable to attack because it improperly validates input from connected devices. Security research in this area has encouraged software vendors to treat connected devices as untrustworthy.

There is no mechanism that will completely prevent this type of attack because data sent from a page to the device is an opaque sequence of bytes. Efforts to block a particular type of data from being sent are likely be met by workarounds on the part

of device manufacturers who nevertheless want to send this type of data to their devices.

User agents can implement additional mechanisms to control access to devices:

- A setting which prevents sites from calling `requestPort()` unless added to an explicit allow list.

  Systems administrators could apply such a setting across their managed fleet using enterprise policy controls. Such controls may allow the administrator to automatically grant selected sites access to particular devices and no others.

- A list of device IDs for hardware which is known to be exploitable could be deployed with the user agent. Connections to listed devices would be blocked. An implementation could use its automatic update or experiment management system to deploy updates to this list on the fly to block an active attack.

Implementations of [WEB-BLUETOOTH] and [WEBUSB] have experimented with these mitigations however there are limits to their effectiveness. First, it is difficult to define whether a device is exploitable. For example, this API will allow a site to upload firmware to a microcontroller development board. This is a key use case for this API as these devices are common in the educational and hobbyist markets. These boards do not implement firmware signature verification and so can easily be turned into a malicious device. These boards are clearly exploitable but should not be blocked.

In addition, maintaining a list of vulnerable devices works well for USB and Bluetooth because those protocols define out-of-band mechanisms to gather device metadata. The make and model of such devices can thus be easily identified even if they present themselves to the host as a virtual serial ports. However, there are generic USB- or Bluetooth-to-serial adapters as well as systems with "real" serial ports using a DB-25, DE-9 or RJ-45 connector. For these there is no metadata that can be read to determine the identity of the device connected to the port and so blocking access to these is not possible.

§ 8. Privacy considerations

This section is non-normative.

Serial ports and serial devices contain two kinds of sensitive information. When a port is a USB or Bluetooth device there are identifiers such as the vendor and product IDs (which identify the make and model) as well as a serial number or MAC address. The serial device itself may also have its own identifier that is available through commands sent via the serial port. The device may also store other private information which may or may not be identifying.

In order to manage device permissions an implementation will likely store device identifiers such as the USB vendor ID, product ID and serial number in its user preferences file to be used as stable identifiers for devices the user has granted sites access to. These would not be shared directly with sites and would be cleared when permission is revoked or site data in general is cleared.

Commands a page can send to the device after it has been granted access a page may also be able to access any of the other sensitive information stored by the device. For the reasons mentioned in 7. Security considerations it is impractical and undesirable to attempt to prevent a page from accessing this information.

Implementations should provide users with complete control over which devices a site can access and not grant device access without user interaction. This is the intention of the `requestPort()` method. This prevents a site from silently enumerating and collecting data from all connected devices. This is similar to the file picker UI. A site has no knowledge of the filesystem, only the files or directories that have been chosen by the user. An implementation could notify the user when a site is using these permissions with an indicator icon appearing in the tab or address bar.

Implementations that provide a "private" or "incognito" browsing mode should ensure that permissions from the user's normal profile do not carry over to such a session and permissions granted in this session are not persisted when the session ends. An implementation may warn the user when granting access to a device in such as session as, similar to entering identifying information by hand, device identifiers and other unique properties available from communicating with the device mentioned previously can be used to identify the user between sessions.

Users may be surprised by the capabilities granted by this API if they do not understand the ways in which granting access to a device breaks traditional isolation boundaries in the web security model. Security UI and documentation should explain that granting a site access to a device could give the site full control over the device and any data contained within.

§ 9. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

§ A. Acknowledgements

The following people contributed to the development of this document.

- Anatol Ulrich
- Chris Mumford
- Clément Menard
- Domenic Denicola
- Dominique Hazael-Massieux
- Florian Loitsch
- Florian Scholz
- Francis Gulotta
- François Beaufort
- Keavon Chambers
- Kenneth Rohde Christiansen
- Marcos Cáceres
- marcoscaceres-remote
- Matt Reynolds
- melhuishj
- Michael Kohler
- Ms2ger
- Reilly Grant
- Rick Waldron

- Sankha Narayan Guria
- Simon Pieters
- Suz Hinton
- Travis Leithead
- Vincent Scheib

§ B. References

§ B.1 Normative references

[dom]
DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/

[html]
HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[infra]
Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[PERMISSIONS-POLICY]
Permissions Policy. Ian Clelland. W3C. 24 July 2024. W3C Working Draft. URL: https://www.w3.org/TR/permissions-policy-1/

[STREAMS]
Streams Standard. Adam Rice; Domenic Denicola; Mattias Buelens; 吉野剛史 (Takeshi Yoshino). WHATWG. Living Standard. URL: https://streams.spec.whatwg.org/

[WEB-BLUETOOTH]
Web Bluetooth. Jeffrey Yasskin. W3C Web Bluetooth Community Group. Draft Community Group Report. URL: https://webbluetoothcg.github.io/web-bluetooth/

[WEBIDL]
Web IDL Standard. Edgar Chen; Timothy Gu. WHATWG. Living Standard.

URL: https://webidl.spec.whatwg.org/

§ B.2 Informative references

[CSP3]

Content Security Policy Level 3. Mike West; Antonio Sartori. W3C. 18 June 2024. W3C Working Draft. URL: https://www.w3.org/TR/CSP3/

[WEBUSB]

WebUSB API. WICG. cg-draft. URL: https://wicg.github.io/webusb/

↥