WebUSB API

Unofficial Draft 21 October 2015

Editors:
Reilly Grant, Google
Ken Rockot, Google

Abstract

This document describes an API for direct access to Universal Serial Bus devices from web pages.

Status of This Document

This document is merely a public working draft of a potential specification. It has no official standing of any kind and does not represent the support or consensus of any standards organisation.

Table of Contents

1. Introduction

This section is non-normative.

Today when you connect a device to your computer you hope that somehow it will find the right driver and it will Just Work™. For lots of devices it does because there are standardized drivers for things like keyboards, mice, hard drives and webcams built into the operating system. What about the long tail of unusual devices or the next generation of gadgets that haven't been standardized yet? WebUSB takes "plug and play" to the next level by connecting devices to the software that drives them across any platform by harnessing the power of web technologies.

2. Security and Privacy Considerations

USB hosts and devices historically trust each other. There are published attacks against USB devices that will accept unsigned firmware updates. These vulnerabilities permit an attacker to gain a foothold in the device and attack the original host or any other host to which they are later connected. For this reason WebUSB does not attempt to provide a mechanism for any web page to connect to arbitrary devices.

Direct access to peripherals also poses a privacy risk. Knowing the make and model of connected devices provides additional bits of entropy for fingerprinting. If devices also posess some form of serial number then they can be uniquely identifying. Additionally a device may have access to data about its environment or directly store user data.

For this reason two checks SHOULD be combined before a site is granted access to a device. First, so that the device can protect itself from malicious sites a set of allowed origins MUST be read from the device (or from a public registry) and checked against the requesting origin. Second, so that the user is protected from malicious sites the UA SHOULD prompt the user for authorization to allow the site to detect the presense of a device and connect to it.

To help ensure that only the entity the user approved for access actually has access, this specification requires that only secure contexts as described in [powerful-features] can access USB devices.

3. Device Requirements

To be supported by a page using this API a USB device MUST provide information to the UA about the origins authorized to connect to it and MAY also provide a landing page that the UA MAY direct the user to navigate to in order to interact with the device.

3.1 *WebUSB Platform Capability Descriptor*

Communication with a device starts with the UA finding the following Platform Descriptor in the device's Binary Object Store:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor. Must be set to 23.
1	bDescriptorType	1	Constant	DEVICE CAPABILITY descriptor type ([USB31] Table 9-6).
2	bDevCapabilityType	1	Constant	PLATFORM capability type ([USB31] Table 9-14).
3	bReserved	1	Number	This field is reserved and shall be set to zero.
4	PlatformCapabilityUUID	16	UUID	Must be set to {3408b638-09a9-47a0-8bfd-a0768815b665}.
20	bcdVersion	2	BCD	Protocol version supported. Must be set to 0x0100.
22	bVendorCode	1	Number	bRequest value used for issuing WebUSB requests.

3.2 WebUSB Device Requests

All control transfers defined by this specification are considered to be vendor-specific requests. The `bVendorCode` value found in the WebUSB Platform Capability Descriptor provides the UA with the `bRequest` the device expects the host to use when issuing control transfers these requests. The request type is then specified in the `wIndex` field.

WebUSB Request Codes

Constant	Value
GET_ALLOWED_ORIGINS	1
GET_LANDING_PAGE	2

3.2.1 Get Allowed Origins

This request gets the set of origins allowed to access the device. The device MUST return a Descriptor Set Header containing one or more URL Descriptors possibly contained within a combination of Configuration Subset Headers and Function Subset Headers which limit their scope to particular portions of the device.

The URLs returned by this request MUST be interpreted as origins (as defined by [RFC6454]) and so content beyond the scheme/host/port triple MAY be ignored.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
11000000B	`bVendorCode`	Zero	GET_ALLOWED_ORIGINS	Descriptor Length	Descriptor

3.2.2 Get Landing Page

This request gets the landing page to which the UA can be navigated in order to interact with the device. While a device may be accessible by multiple origins it MUST only have a single landing page or none at all.

The data returned MUST be a single URL Descriptor.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
11000000B	`bVendorCode`	Zero	GET_LANDING_PAGE	Descriptor Length	Descriptor

3.3 WebUSB Descriptors

These descriptor types are returned by requests defined in this specification.

WebUSB Descriptor Types

Constant	Value
WEBUSB_DESCRIPTOR_SET_HEADER	0

WEBUSB_CONFIGURATION_SUBSET_HEADER	1
WEBUSB_FUNCTION_SUBSET_HEADER	2
WEBUSB_URL	3

3.3.1 *Descriptor Set Header*

A response referring to multiple origins MUST begin with this header to identify the total length of the data to follow.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor. Must be set to 4.
1	bDescriptorType	1	Constant	WEBUSB_DESCRIPTOR_SET_HEADER.
2	wTotalLength	2	Number	Total size of this and all following descriptors.

3.3.2 *Configuration Subset Header*

This header declares that the descriptors following it (up to `wTotalLength` bytes) are scoped to the USB device configuration described by the configuration descriptor with the given `bConfigurationValue`.

This descriptor MUST be contained within a Descriptor Set Header.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor. Must be set to 5.
1	bDescriptorType	1	Constant	WEBUSB_CONFIGURATION_SUBSET_HEADER.
2	bConfigurationValue	1	Number	Configuration to which this section applies.
3	wTotalLength	2	Number	Total size of this and the following descriptors to which this header applies.

3.3.3 *Function Subset Header*

This header declares that the descriptors following it (up to `wTotalLength` bytes) are scoped to the USB device configuration described by the configuration descriptor with the given `bConfigurationValue`.

This descriptor MUST be contained within a Descriptor Set Header.

WebUSB Function Subset Header

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor. Must be set to 5.
1	bDescriptorType	1	Constant	WEBUSB_FUNCTION_SUBSET_HEADER.
2	bFirstInterfaceNumber	1	Number	First interface of the function to which this section applies.
3	wTotalLength	2	Number	Total size of this and the following descriptors to which this header applies.

3.3.4 *URL Descriptor*

This descriptor contains a single URL. It may be contained within a Descriptor Set Header to apply to the entire device, a Configuration Subset Header to apply to a specific configuration, or a Function Subset Header to apply to the set of interfaces comprising that function.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor.
1	bDescriptorType	1	Constant	WEBUSB_URL.
2	URL	Variable	String	UTF-8 encoded URL.

3.4 Public Device Registry

The WebUSB Platform Capability Descriptor and descriptors returned by the requests defined above can be elided by publishing this information in a public registry of supported USB devices. This will allow device manufacturers to support WebUSB on existing devices.

4. Device Enumeration

WebIDL

```
dictionary USBDeviceFilter {
    unsigned short vendorId;
    unsigned short productId;
    octet          classCode;
```

```
    octet          subclassCode;
    octet          protocolCode;
};

dictionary USBDeviceRequestOptions {
    required sequence<USBDeviceFilter> filters;
};

[NoInterfaceObject]
interface USB {
                attribute EventHandler onconnect;
                attribute EventHandler ondisconnect;
    Promise<sequence<USBDevice>> getDevices();
    Promise<USBDevice>           requestDevice(USBDeviceRequestOptions options);
};

USB implements EventTarget;

partial interface Navigator {
    readonly    attribute USB usb;
};
```

The ***vendorId*** and ***productId*** field will cause the filter to match any device with the given vendor and (optionally) product identifiers.

The ***classCode***, ***subclassCode*** and ***protocolCode*** fields will cause the filter to match any device that implements the given class, class and subclass, or class, subclass and protocol tuple and any composite device with an interface implementing the same. A subclass MUST NOT be specified unless a class is provided and a protocol MUST NOT be specified unless a subclass is also provided.

The UA MUST be able to ***enumerate all devices attached to the system***. It is, however NOT required to perform this work each time an algorithm requests an enumeration. The UA MAY cache the result of the first enumeration it performs and then begin monitoring for device connection and disconnection events, adding connected devices to its cached enumeration and removing disconnected devices. This mode of operation is preferred as it reduces the number of operating system calls made and amount of bus traffic generated by the `getDevices()` and `requestDevice()` methods.

The UA MUST maintain an ***allowed devices set*** for each script execution environment. Once a device is added to this set it SHALL remain in the set for a period of time determined by the UA's ability to identify the device.

- For a device with a unique identifier such as a serial number or container ID the device SHALL remain in the allowed devices set until explicitly removed by the user. Vendor and product IDs MUST NOT be considered uniquely identifying.
- For a device without a unique identifier the device SHALL remain in the allowed devices set until it becomes uncertain whether the device connected to the host is still the device originally added to the set. This MAY happen when the device is disconnected from the host, when the UA exits or, if tracked by the host operating system, when the host is shut down.

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

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

The ***getDevices()*** method, when invoked, MUST return a new promise and run the following steps in parallel:

1. If the incumbent settings object is not a secure context, reject *promise* with a `SecurityError` and abort these steps.
2. Enumerate all devices attached to the system. Let this result be *enumerationResult*.
3. Remove all devices from *enumerationResult* that are not in the current script execution environment's allowed devices set.
4. For each remaining *device* in *enumerationResult* get the `USBDevice` object representing *device*, and add the result to *devices*.
5. Resolve *promise* with *devices*.

The ***requestDevice(options)*** method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the incumbent settings object is not a secure context, reject *promise* with a `SecurityError` and abort these steps.
2. If the algorithm is not allowed to show a popup, reject *promise* with a `SecurityError` and abort these steps.
3. Enumerate all devices attached to the system. Let this result be *enumerationResult*.
4. Remove all devices from *enumerationResult* that do not match at least one of the filters in *options*.`filters`.

   The UA SHOULD apply additional origin-based filtering of available devices by consulting an authoritative list of device-origin mappings or referring to the origin list specified in the device's Binary Object Store.

   The UA MAY provide additional mechanisms for blacklisting or whitelisting specific devices for arbitrary origins.

5. Even if *enumerationResult* is empty, display a prompt to the user requesting that the user select a device from it. The UA SHOULD show a human-readable name of each device.
6. Wait for the user to have selected a *device* or cancelled the prompt.
7. If the user cancels the prompt, reject *promise* with a `NotFoundError` and abort these steps.
8. Add *device* to the current script execution environment's allowed devices set.
9. Get the `USBDevice` object representing *device* and resolve *promise* with that object.

4.1 Events

WebIDL

```
interface USBConnectionEvent : Event {
```

```
    readonly    attribute USBDevice device;
};
```

When the UA detects a new USB device connected to the host it MUST perform the following steps for each script execution environment:

1. Let *device* be the `USBDevice` object representing the device.
2. If *device* is not in the allowed devices set for the current script execution environment abort these steps.
3. Let *event* be a new `USBConnectionEvent`, with the `device` attribute set to *device*.
4. Fire an event named ***connect*** on `navigator.usb`, using *event* as the event object.

When the UA detects a previously connected USB device has been disconnected from the host it MUST perform the following steps for each script execution environment:

1. Let *device* be the `USBDevice` object representing the device.
2. If *device* is not in the allowed devices set for the current script execution environment abort these steps.
3. Let *event* be a new `USBConnectionEvent`, with the `device` attribute set to *device*.
4. Fire an event named ***disconnect*** on `navigator.usb`, using *event* as the event object.
5. Consider removing *device* from the allowed devices set.

5. Device Usage

WebIDL

```
interface USBDevice {
    readonly    attribute DOMString                    guid;
    readonly    attribute octet                        usbVersionMajor;
    readonly    attribute octet                        usbVersionMinor;
    readonly    attribute octet                        usbVersionSubminor;
    readonly    attribute octet                        deviceClass;
    readonly    attribute octet                        deviceSubclass;
    readonly    attribute octet                        deviceProtocol;
    readonly    attribute unsigned short               vendorId;
    readonly    attribute unsigned short               productId;
    readonly    attribute octet                        deviceVersionMajor;
    readonly    attribute octet                        deviceVersionMinor;
    readonly    attribute octet                        deviceVersionSubminor;
    readonly    attribute DOMString?                   manufacturerName;
    readonly    attribute DOMString?                   productName;
    readonly    attribute DOMString?                   serialNumber;
    readonly    attribute FrozenArray<USBConfiguration> configurations;
    Promise<void>                 open();
    Promise<void>                 close();
    Promise<USBConfiguration>     getConfiguration();
    Promise<void>                 setConfiguration(octet configurationValue);
    Promise<void>                 claimInterface(octet interfaceNumber);
    Promise<void>                 releaseInterface(octet interfaceNumber);
    Promise<void>                 setInterface(octet interfaceNumber, octet alternateSetting);
    Promise<USBInTransferResult>  controlTransferIn(USBControlTransferParameters setup, unsigned short length);
    Promise<USBOutTransferResult> controlTransferOut(USBControlTransferParameters setup, optional BufferSource data);
    Promise<void>                 clearHalt(octet endpointNumber);
    Promise<USBInTransferResult>  transferIn(octet endpointNumber, unsigned long length);
    Promise<USBOutTransferResult> transferOut(octet endpointNumber, BufferSource data);
    Promise<void>                 reset();
};
```

The ***guid*** attribute indicates a unique identifier string for the device. This identifier SHALL remain consistent for the lifetime of a device's connection to the USB host.

The ***usbVersionMajor***, ***usbVersionMinor*** and ***usbVersionSubminor*** attributes declare the USB protocol version supported by the device. They SHALL correspond to the value of the `bcdUSB` field of the device descriptor such that a value of `0xJJMN` has major version `JJ`, minor version `M` and subminor version `N`.

The ***deviceClass***, ***deviceSubclass*** and ***deviceProtocol*** attributes declare the communication interface supported by the device. They MUST correspond respectively to the values of the `bDeviceClass`, `bDeviceSubClass` and `bDeviceProtocol` fields of the device descriptor.

The ***vendorId*** and ***productId*** attribute declares the vendor ID of the device manufacturer and product ID assigned by the device manufacturer. They SHALL correspond to the values of the `idVendor` and `idProduct` fields of the device descriptor.

The ***deviceVersionMajor***, ***deviceVersionMinor*** and ***deviceVersionSubminor*** attributes declare the device release number as defined by the device manufacturer. It SHALL correspond to the value of the `bcdDevice` field of the device descriptor such that a value of `0xJJMN` has major version `JJ`, minor version `M` and subminor version `N`.

The ***configurations*** attribute contains a list of configurations supported by the device. These configurations SHALL be populated from the configuration descriptors reported by the device and the number of elements in this list SHALL match the value of the `bNumConfigurations` field of the device descriptor.

The ***manufacturerName***, ***productName*** and ***serialNumber*** attributes SHOULD contain the values of the string descriptors referenced by the

`iManufacturer`, `iProduct` and `iSerialNumber` fields of the device descriptor if each is available.

The ***open***() method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. If the device is already in the open state, reject *promise* with a `InvalidStateError` and abort these steps.
3. Perform the necessary platform-specific steps to begin a session with the device. If these fail for any reason reject *promise* with a `NetworkError` and abort these steps.
4. Consider the device to be in what will be referred to as the ***open state*** and resolve *promise*.

The ***close***() method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. If the device is not in the open state, reject *promise* with a `InvalidStateError` and abort these steps.
3. Abort all other algorithms currently running against this device and reject their associated promises with an `AbortError`.
4. Perform the necessary platform-specific steps to release any claimed interfaces as if `releaseInterface(`*interfaceNumber*`)` had been called for each claimed interface.
5. Perform the necessary platform-specific steps to end the session with the device.
6. Take the device out of the open state and resolve *promise*.

The ***getConfiguration***() method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *configuration* be the active device configuration. If the device is unconfigured, reject *promise* with a `NotFoundError` and abort these steps.
3. Resolve *promise* with *configuration*.

The ***setConfiguration***(*configurationValue*) method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *configuration* be the device configuration with `bConfigurationValue` equal to *configurationValue*. If no such configuration exists, reject *promise* with a `NotFoundError` and abort these steps.
3. If the device is not in the open state, reject *promise* with a `InvalidStateError` and abort these steps.
4. Abort all transfers currently scheduled on endpoints other than the default control pipe and reject their associated promises with a `AbortError`.
5. Issue a `SET_CONFIGURATION` control transfer to the device to set *configurationValue* as its active configuration. If this step fails reject *promise* with a `NetworkError` and abort these steps.
6. Resolve *promise*.

The ***claimInterface***(*interfaceNumber*) method, when invoked, MUST return a new promise and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *interface* be the interface in the active configuration with `bInterfaceNumber` equal to *interfaceNumber*. If no such interface exists, reject *promise* with a `NotFoundError` and abort these steps.
3. If the device is not in the open state or *interface* is already in the claimed state, reject *promise* with an `InvalidStateError` and abort these steps.
4. If the requester is not allowed to access *interface*, reject *promise* with a `SecurityError` and abort these steps.
5. Perform the necessary platform-specific steps to request exclusive control over *interface*. If this fails, reject *promise* with a `NetworkError` and abort these steps.
6. Consider *interface* to be in what will be referred to as the ***claimed state*** and resolve *promise*.

The ***releaseInterface***(*interfaceNumber*) method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *interface* be the interface in the active configuration with `bInterfaceNumber` equal to *interfaceNumber*. If no such interface exists, reject *promise* with a `NotFoundError` and abort these steps.
3. If the device is not in the open state or *interface* is not in the claimed state, reject *promise* with an `InvalidStateError` and abort these steps.
4. Perform the necessary platform-specific steps to reliquish exclusive control over *interface*.
5. Take *interface* out of the claimed state and resolve *promise*.

The ***setInterface***(*interfaceNumber, alternateSetting*) method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *interface* be the interface in the active configuration with `bInterfaceNumber` equal to *interfaceNumber*. If no such interface exists, reject *promise* with a `NotFoundError` and abort these steps.
3. If the device is not in the open state or *interface* is not in the claimed state, reject *promise* with an `InvalidStateError` and abort these steps.
4. Abort all transfers currently scheduled on endpoints associated with the previously selected alternate setting of *interface* and reject their associated promises with a `AbortError`.
5. Issue a `SET_INTERFACE` control transfer to the device to set *alternateSetting* as the current configuration of *interface*. If this step fails reject *promise* with a `NetworkError` and abort these steps.
6. Resolve *promise*.

The ***controlTransferIn***(*setup, length*) method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. If the device is not in the open state, reject *promise* with an `InvalidStateError` and abort these steps.
3. Check the validity of the control transfer parameters and abort these steps if *promise* is rejected.
4. If *length* is greater than the `wMaxPacketSize0` field of the device's device descriptor, reject *promise* with a `TypeError` and abort these steps.
5. Let *result* be a new `USBInTransferResult` and let *buffer* be a new `ArrayBuffer` of *length* bytes.
6. Issue a control transfer with the setup packet parameters provided in *setup* and the data transfer direction in `bmRequestType` set to "device to host" and `wLength` set to *length*.
7. If the device responds with data, store the first *length* bytes of this data in *buffer* and set *result.data* to *buffer*.
8. If the device responds by stalling the default control pipe set *result.status* to `"stall"`.
9. If more than *length* bytes are received set *result.status* to `"babble"` and otherwise set it to `"ok"`.
10. If the transfer fails for any other reason reject *promise* with a `NetworkError` and abort these steps.
11. Resolve *promise* with *result*.

The ***controlTransferOut***(*setup, data*) method, when invoked, must return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. If the device is not in the open state, reject *promise* with an `InvalidStateError` and abort these steps.
3. Check the validity of the control transfer parameters and abort these steps if *promise* is rejected.
4. If *data.length* is greater than the `wMaxPacketSize0` field of the device's device descriptor, reject *promise* with a `TypeError` and abort these steps.
5. Issue a control transfer with the setup packet populated by *setup* and the data transfer direction in `bmRequestType` set to "host to device" and `wLength` set to *data.length*. Transmit *data* in the data stage of the transfer.
6. Let *result* be a new `USBOutTransferResult`.
7. If the device responds by stalling the default control pipe set *result.status* to `"stall"`.
8. If the device acknowledges the transfer set *result.status* to `"ok"` and *result.bytesWritten* to *data.length*.
9. If the transfer fails for any other reason reject *promise* with a `NetworkError` and abort these steps.
10. Resolve *promise* with *result*.

The ***clearHalt***(*direction, endpointNumber*) method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *endpoint* be the endpoint in the active configuration with `bEndpointAddress` corresponding to *direction* and *endpointNumber*. If no such endpoint exists reject *promise* and abort these steps.
3. If the device is not in the open state or the interface containing *endpoint* is not in the claimed state, reject *promise* with an `InvalidStateError` and abort these steps.
4. Issue a `CLEAR_FEATURE` control transfer to the device to clear the stall condition on *endpoint*.
5. On failure reject *promise* with a `NetworkError`, otherwise resolve *promise*.

The ***transferIn***(*endpointNumber, length*) method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *endpoint* be the IN endpoint in the active configuration with `bEndpointAddress` corresponding to *endpointNumber*. If there is no such endpoint reject *promise* with a `NotFoundError` and abort these steps.
3. If the device is not in the open state or the interface containing *endpoint* is not in the claimed state, reject *promise* with an `InvalidStateError` and abort these steps.
4. As appropriate for *endpoint* enqueue a bulk or interrupt IN transfer on *endpoint* with a buffer sufficient to receive *length* bytes of data from the device.
5. Let *result* be a new `USBInTransferResult`.
6. If data is returned as part of this transfer let *buffer* be a new ArrayBuffer of exactly the length of the data received and set *result.data* to *buffer*.
7. If the device responds with more than *length* bytes of data set *result.status* to `"babble"`.
8. If the transfer ends because *endpoint* is stalled set *result.status* to `"stall"`.
9. If the device acknowledges the complete transfer set *result.status* to `"ok"`.
10. If the transfer fails for any other reason reject *promise* with a `NetworkError` and abort these steps.
11. Resolve *promise* with *result*.

The ***transferOut***(*endpointNumber, data*) method, when invoked, MUST return a new promise *promise* and run the following steps in parallel:

1. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
2. Let *endpoint* be the OUT endpoint in the active configuration with `bEndpointAddress` corresponding to *endpointNumber*. If there is no such endpoint reject *promise* with a `NotFoundError` and abort these steps.
3. If the device is not in the open state or the interface containing *endpoint* is not in the claimed state, reject *promise* with an `InvalidStateError` and abort these steps.
4. As appropriate for *endpoint* enqueue a bulk or interrupt OUT transfer on *endpoint* to transmit *data* to the device.
5. Let *result* be a new `USBOutTransferResult`.
6. Set *result.bytesWritten* to the amount of data successfully sent to the device.
7. If the endpoint is stalled set *result.status* to `"stall"`.
8. If the device acknowledges the complete transfer set *result.status* to `"ok"`.

9. If the transfer fails for any other reason reject *promise* with a `NetworkError` and abort these steps.
10. Resolve *promise* with *result*.

The ***`reset()`*** method, when invoked, must return a new promise *promise* and run the following steps in parallel:

1. If the device is not in the open state, reject *promise* with an `InvalidStateError` and abort these steps.
2. If the device is no longer connected to the system, reject *promise* with a `NotFoundError` and abort these steps.
3. Abort all operations on the device and reject their associated promises with an `AbortError`.
4. Perform the necessary platform-specific operation to soft reset the device.
5. On failure reject *promise* with a `NetworkError`, otherwise resolve *promise*.

> ISSUE 1
>
> What configuration is the device in after it resets?

5.1 Transfers

WebIDL

```
enum USBRequestType {
    "standard",
    "class",
    "vendor"
};

enum USBRecipient {
    "device",
    "interface",
    "endpoint",
    "other"
};

enum USBTransferStatus {
    "ok",
    "stall",
    "babble"
};

dictionary USBControlTransferParameters {
    required USBRequestType requestType;
    required USBRecipient   recipient;
    required octet          request;
    required unsigned short value;
    required unsigned short index;
};

interface USBInTransferResult {
    readonly    attribute ArrayBuffer       data;
    readonly    attribute USBTransferStatus status;
};

interface USBOutTransferResult {
    readonly    attribute unsigned long     bytesWritten;
    readonly    attribute USBTransferStatus status;
};
```

A ***control transfer*** is a special class of USB traffic most commonly used for configuring a device. It consists of three stages: setup, data and status. In the ***setup stage*** a ***setup packet*** is transmitted to the device containing request parameters including the transfer direction and size of the data to follow. In the ***data stage*** that data is either sent to or received from the device. In the ***status stage*** successful handling of the request is acknowledged or a failure is signaled.

All USB devices `MUST` have a ***default control pipe*** which is *endpointNumber* `0`.

The ***`requestType`*** attribute populates part of the `bmRequestType` field of the setup packet to indicate whether this request is part of the USB standard, a particular USB device class specification or a vendor-specific protocol.

The ***`recipient`*** attribute populates part of the `bmRequestType` field of the setup packet to indicate whether the control transfer is addressed to the entire device, or a specific interface or endpoint.

The ***`request`*** attribute populates the `bRequest` field of the setup packet. Valid requests are defined by the USB standard, USB device class specifications or the device vendor.

The ***`value`*** and ***`index`*** attributes populate the `wValue` and `wIndex` fields of the setup packet respectively. The meaning of these fields depends on the request being made.

To ***check the validity of the control transfer parameters*** perform the following steps:

1. Let *setup* be the `USBControlTransferParameters` created for the transfer.
2. Let *promise* be the promise created for the transfer.

3. If *setup.recipient* is `"interface"`, perform the following steps:
    1. Let *interfaceNumber* be the lower 8 bits of *setup.wIndex*.
    2. Let *interface* be the interface in the active configuration with `bInterfaceNumber` equal to *interfaceNumber*. If no such interface exists, reject *promise* with a `NotFoundError` and abort these steps.
    3. If *interface* is not in the claimed state, reject *promise* with an `InvalidStateError`.
4. If *setup.recipient* is `"endpoint"`, run the following steps:
    1. Let *endpointNumber* be defined as the lower 4 bits of *setup.wIndex*.
    2. Let *direction* be defined as `"in"` if the 8th bit of *setup.wIndex* is `1` and `"out"` otherwise.
    3. Let *endpoint* be the endpoint in the active configuration with `bEndpointAddress` corresponding to *direction* and *endpointNumber*. If no such endpoint exists, reject *promise* with a `NotFoundError` and abort these steps.
    4. If the interface in which *endpoint* is defined is not in the claimed state, reject *promise* with an `InvalidStateError`.

5.2 Configurations

WebIDL

```
[Constructor(USBDevice device, octet configurationValue)]
interface USBConfiguration {
    readonly    attribute octet                     configurationValue;
    readonly    attribute DOMString?                configurationName;
    readonly    attribute FrozenArray<USBInterface> interfaces;
};
```

Each device configuration SHALL have a unique ***configurationValue*** that matches the `bConfigurationValue` fields of the configuration descriptor that defines it.

The ***configurationName*** attribute SHOULD contain the value of the string descriptor referenced by the `iConfiguration` field of the configuration descriptor, if available.

The ***interfaces*** attribute SHALL contain a list of interfaces exposed by this device configuration. These interfaces SHALL by populated from the interface descriptors contained within this configuration descriptor.

> ISSUE 2
>
> Include some non-normative information about device configurations

5.3 Interfaces

WebIDL

```
[Constructor(USBConfiguration configuration, octet interfaceNumber)]
interface USBInterface {
    readonly    attribute octet                               interfaceNumber;
    readonly    attribute FrozenArray<USBAlternateInterface> alternates;
};

[Constructor(USBInterface deviceInterface, octet alternateSetting)]
interface USBAlternateInterface {
    readonly    attribute octet                     alternateSetting;
    readonly    attribute octet                     interfaceClass;
    readonly    attribute octet                     interfaceSubclass;
    readonly    attribute octet                     interfaceProtocol;
    readonly    attribute DOMString?                interfaceName;
    readonly    attribute FrozenArray<USBEndpoint> endpoints;
};
```

Each interface provides a collection of ***alternates*** identified by a single `bInterfaceNumber` field found in their interface descriptors. The ***interfaceNumber*** attribute MUST match this field.

Each alternative interface configuration SHALL have a unique ***alternateSetting*** within a given interface that matches the `bAlternateSetting` field of the interface descriptor that defines it.

The ***interfaceClass***, ***interfaceSubclass*** and ***interfaceProtocol*** attributes declare the communication interface supported by the interface. They MUST correspond respectively to the values of the `bInterfaceClass`, `bInterfaceSubClass` and `bInterfaceProtocol` fields of the interface descriptor.

The ***interfaceName*** attribute SHOULD contain the value of the string descriptor referenced by the `iInterface` field of the interface descriptor, if available.

The ***endpoints*** attribute SHALL contain a list of endpoints exposed by this interface. These endpoints SHALL by populated from the endpoint descriptors contained within this interface descriptor and the number of elements in this sequence SHALL match the value of the `bNumEndpoints` field of the interface descriptor.

A device's ***active configuration*** is the combination of the `USBConfiguration` selected by calling `setConfiguration(configurationValue)` and the set of `UsbAlternateInterface`s selected by calling `setInterface(interfaceNumber,`

alternateSetting). A device MAY, by default, be left in an unconfigured state, referred to as configuration 0 or may automatically be set to whatever configuration has bConfigurationValue equal to 1. When a configuration is set all interfaces within that configuration automatically have the USBAlternateInterface with bAlternateSetting equal to 0 selected by default. It is therefore unnecessary to call setInterface(*interfaceNumber*, 0) for each interface when opening a device.

5.4 Endpoints

WebIDL

```
enum USBDirection {
    "in",
    "out"
};

enum USBEndpointType {
    "bulk",
    "interrupt",
    "isochronous"
};

[Constructor(USBAlternateInterface alternate, octet endpointNumber, USBDirection direction)]
interface USBEndpoint {
    readonly    attribute octet           endpointNumber;
    readonly    attribute USBDirection    direction;
    readonly    attribute USBEndpointType type;
    readonly    attribute unsigned long   packetSize;
};
```

Each endpoint within a particular device configuration SHALL have a unique combination of endpointNumber and direction. The ***endpointNumber*** MUST equal the 4 least significant bits of the bEndpointAddress field of the endpoint descriptor defining the endpoint.

The ***direction*** attribute declares the transfer direction supported by this endpoint and is equal to "in" if the most significant bit of the bEndpointAddress is set and "out" otherwise. An endpoint may either carry data IN from the device to host or OUT from host to device.

The ***type*** attribute declares the type of data transfer supported by this endpoint.

The ***packetSize*** attribute declares the packet size employed by this endpoint and MUST be equal to the value of the wMaxPacketSize of the endpoint descriptor defining it. In a High-Speed, High-Bandwidth endpoint this value will include the multiplication factor provided by issuing multiple transactions per microframe. In a SuperSpeed device this value will include the multiplication factor provided by the bMaxBurst field of the SuperSpeed Endpoint Companion descriptor.

6. Terminology

This specification uses several terms taken from [USB31]. While reference is made to version 3.1 of the Universal Serial Bus many of these concepts exist in previous versions as well. Significant differences between USB versions that have barring on this specification will be called out explicitly.

Descriptors are binary data structures that can be read from a device and describe its properties and function:

- The ***device descriptor*** contains information applicable to the entire devices and is described in section 9.6.1 of [USB31].
- A ***configuration descriptor*** describes a particular set of device interfaces and endpoints that can be selected by the host. Its fields are described in section 9.6.3 of [USB31].
- An ***interface descriptor*** describes the interface of a particular functional component of a device including its protocol and communication endpoints. Its fields are described in section 9.6.5 of [USB31].
- An ***endpoint descriptor*** describes a channel through which data is either sent to or received from the device. Its fields are described in section 9.6.6 of [USB31].

The ***Binary Object Store*** (BOS) is an additional set of descriptors that are more free-form than the standard device descriptors. Of note is the ***Platform Descriptor*** type which allows third parties (such as this specification) to declare their own types of descriptors. Each of these is identified by a UUID. The Binary Object Store is described in section 9.6.2 of [USB31].

A. References

A.1 Informative references

[RFC6454]
A. Barth. *The Web Origin Concept*. December 2011. Proposed Standard. URL: https://tools.ietf.org/html/rfc6454

[USB31]
Universal Serial Bus 3.1 Specification. 26 July 2013. URL: http://www.usb.org/developers/docs/usb_31_060115.zip

[powerful-features]
Mike West; Yan Zhu. *Privileged Contexts*. 24 April 2015. W3C Working Draft. URL: http://www.w3.org/TR/powerful-features/