W3C Working Draft

Web MIDI API

W3C Working Draft 21 January 2025

▼ More details about this document

This version:
https://www.w3.org/TR/2025/WD-webmidi-20250121/

Latest published version:
https://www.w3.org/TR/webmidi/

Latest editor's draft:
https://webaudio.github.io/web-midi-api/

History:
https://www.w3.org/standards/history/webmidi/
Commit history

Test suite:
https://github.com/web-platform-tests/wpt/tree/master/webmidi

Editors:
Chris Wilson (Google)
Michael Wilson (Google)

Former editor:
Jussi Kalliokoski

Feedback:
GitHub WebAudio/web-midi-api (pull requests, new issue, open issues)

Implementation report:
No preliminary interoperability or implementation report exists.

Abstract

Some user agents have music devices, such as synthesizers, keyboard and other controllers, and drum machines connected to their host computer or device. The widely adopted Musical Instrument Digital Interface (MIDI) protocol enables electronic musical instruments, controllers and computers to communicate and synchronize with each other. MIDI does not transmit audio signals: instead, it sends

event messages about musical notes, controller signals for parameters such as volume, vibrato and panning, cues and clock signals to set the tempo, and system-specific MIDI communications (e.g. to remotely store synthesizer-specific patch data). This same protocol has become a standard for non-musical uses, such as show control, lighting and special effects control.

This specification defines an API supporting the MIDI protocol, enabling web applications to enumerate and select MIDI input and output devices on the client system and send and receive MIDI messages. It is intended to enable non-music MIDI applications as well as music ones, by providing low-level access to the MIDI devices available on the users' systems. The Web MIDI API is not intended to describe music or controller inputs semantically; it is designed to expose the mechanics of MIDI input and output interfaces, and the practical aspects of sending and receiving MIDI messages, without identifying what those actions might mean semantically (e.g., in terms of "modulate the vibrato by 20Hz" or "play a G#7 chord", other than in terms of changing a controller value or sending a set of note-on messages that happen to represent a G#7 chord).

To some users, "MIDI" has become synonymous with Standard MIDI Files and General MIDI. That is not the intent of this API; the use case of simply playing back a .SMF file is not within the purview of this specification (it could be considered a different format to be supported by the HTML `audio` element, for example). The Web MIDI API is intended to enable direct access to devices that respond to MIDI - controllers, external synthesizers or lighting systems, for example. The Web MIDI API is also explicitly designed to enable a new class of applications on the web that can respond to MIDI controller inputs - using external hardware controllers with physical buttons, knobs and sliders (as well as musical controllers like keyboard, guitar or wind instrument controllers) to control web applications.

The Web MIDI API is also expected to be used in conjunction with other APIs and elements of the web platform, notably the Web Audio API. This API is also intended to be familiar to users of MIDI APIs on other systems, such as Apple's CoreMIDI and Microsoft's Windows MIDI API.

Status of This Document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the Audio Working Group as a Working Draft using the Recommendation track.

Publication as a Working Draft does not imply endorsement by W3C and its Members.

Table of Contents

§ 1. Introduction

This section is non-normative.

The Web MIDI API specification defines a means for web developers to enumerate, manipulate and access MIDI devices - for example, interfaces that may provide hardware MIDI ports with other devices plugged in to them and USB devices that support the USB-MIDI specification. Having a Web API for MIDI enables web applications that use existing software and hardware synthesizers, hardware music controllers and light systems and other mechanical apparatus controlled by MIDI. This API has been defined with this wide variety of use cases in mind.

The approaches taken by this API are similar to those taken in Apple's CoreMIDI API and Microsoft's Windows MIDI API; that is, the API is designed to represent the low-level software protocol of MIDI, in order to enable developers to build powerful MIDI software on top. The API enables the developer to enumerate input and output interfaces, and send and receive MIDI messages, but (similar to the aforementioned APIs) it does not attempt to semantically define or interpret MIDI messages beyond what is necessary to robustly support current devices.

The Web MIDI API is not intended to directly implement high-level concepts such as sequencing; it does not directly support Standard MIDI Files, for example, although a Standard MIDI File player can be built on top of the Web MIDI API. It is also not intended to semantically capture patches or controller assignments, as General MIDI does; such interpretation is outside the scope of the Web MIDI API (though again, General MIDI can easily be utilized through the Web MIDI API).

§ 2. 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.

The key words *MUST* and *SHOULD* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification defines conformance criteria that apply to a single product: the user agent that implements the interfaces that it contains.

Implementations that use ECMAScript to implement the APIs defined in this specification *MUST* implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [WEBIDL], as this specification uses that specification and terminology.

§ 3. Terminology

The ***Web Audio API*** and its associated interfaces and concepts are defined in [webaudio].

The terms ***MIDI***, ***MIDI device***, ***MIDI input port***, ***MIDI output port***, ***MIDI interface***, ***MIDI message***, ***System Real Time*** and ***System Exclusive*** are defined in [MIDI].

NOTE

Valid MIDI message is defined in [MIDI]. The following may be used as a non-normative guide:

- The first byte (the status byte) should have the high bit set, any following bytes should not unless they are part of a System Exclusive message
- If the high nibble of the status byte in hex is `8`, `9`, `A`, `B`, or `E` then the total message length should be 3 bytes
- If the high nibble of the status byte is `C` or `D` then the total message length should be 2 bytes
- If the status byte is `F1` or `F3` then the total message length should be 2 bytes
- If the status byte is `F2` then the total message length should be 3 bytes
- If the status byte is `F6`, `F8`, `FA`, `FB`, `FC`, `FE`, or `FF` then the total message length should be 1 byte (only the status byte)
- If the status byte is `F0` then this is a System Exclusive message with no length restriction, and the last byte should be `F7`
- If the status byte is `F4`, `F5`, `F7`, `F9`, or `FD` then the message is not valid

§ 4. Obtaining Access to MIDI Devices

§ 4.1 Permissions Integration

The Web Midi API is a powerful feature that is identified by the name "midi". It integrates with *Permissions* by defining the following permission-related flags:

permission descriptor type

WebIDL
```
dictionary MidiPermissionDescriptor : PermissionDescriptor {
  boolean sysex = false;
};
```

`{name: "midi", sysex: true}` is stronger than `{name: "midi", sysex: false}`.

§ 4.2 Permissions Policy Integration

The Web Midi API defines a policy-controlled feature named *"**midi**"* which has a default allowlist of `'self'`.

§ 4.3 Extensions to the `Navigator` interface

```
WebIDL
partial interface Navigator {
  [SecureContext]
  Promise <MIDIAccess> requestMIDIAccess(optional MIDIOptions options =
{});
};
```

***`requestMIDIAccess()`* method**

When invoked, returns a Promise object representing a request for access to MIDI devices on the user's system.

Requesting MIDI access *SHOULD* prompt the user for access to MIDI devices, particularly if System Exclusive access is requested. In some scenarios, this permission may have already been implicitly or explicitly granted, in which case this prompt may not appear. If the user gives express permission or the call is otherwise approved, the vended Promise is resolved. The underlying system may choose to allow the user to select specific MIDI interfaces to expose to this API (i.e. pick and choose interfaces on an individual basis), although this is not required. The system may also choose to prompt (or not) based on whether System Exclusive support is requested, as System Exclusive access has greater privacy and security implications.

If the user declines or the call is denied for any other reason, the Promise is rejected with a `DOMException` parameter.

Multiple calls to `requestMIDIAccess()` are allowed even if not all vended Promises have settled.

When the `requestMIDIAccess()` method is called, the user agent *MUST* run the following steps:

1. Let *promise* be a new Promise object and *resolver* be its associated resolver.

2. Return *promise* and run the following steps asynchronously.

3. Let *document* be the calling context's Document.

4. If *document* is not allowed to use the policy-controlled feature named midi, jump to the step labeled *failure* below.

5. Optionally, e.g. based on a previously-established user preference, for security reasons, or due to platform limitations, jump to the step labeled *failure* below.

6. Optionally, e.g. based on a previously-established user preference, jump to the step labeled *success* below.

7. Prompt the user in a user-agent-specific manner for permission to provide the entry script's origin with a `MIDIAccess` object representing control over user's MIDI devices. This prompt may be contingent upon whether System Exclusive support was requested, and may allow the user to enable or disable that access.

   If permission is denied, jump to the step labeled *failure* below. If the user never responds, this algorithm will never progress beyond this step. If permission is granted, continue the following steps.

8. ***success***: Let *access* be a new `MIDIAccess` object. (It is possible to call requestMIDIAccess() multiple times; this may prompt the user multiple times, so it may not be best practice, and the same instance of MIDIAccess will not be returned each time.)

9. Call *resolver*'s `accept(value)` method with *access* as value argument.

10. Terminate these steps.

11. ***failure***: Let *error* be a new `DOMException`. This exception's .name should be "`NotAllowedError`" if the user or their security settings denied the application from creating a MIDIAccess instance with the requested options, or if the error is the result of *document* not being allowed to use the feature, "`AbortError`" if the page is going to be closed for a user navigation, "`InvalidStateError`" if the underlying systems raise any errors, or otherwise it should be "`NotSupportedError`".

12. Call *resolver*'s `reject(value)` method with *error* as value argument.

§ 4.3.1 *`MIDIOptions`* Dictionary

This dictionary contains optional settings that may be provided to the `requestMIDIAccess()` request.

WebIDL
```
dictionary MIDIOptions {
  boolean sysex;
  boolean software;
};
```

`sysex`

This member informs the system whether the ability to send and receive System Exclusive messages is requested or allowed on a given `MIDIAccess` object. On the option passed to `requestMIDIAccess()`, if this member is set to true, but System Exclusive support is denied (either by policy or by user action), the access request will fail with a "`NotAllowedError`" error. If this support is not requested (and allowed), the system will throw exceptions if the user tries to send System Exclusive messages, and will silently mask out any System Exclusive messages received on the port.

`software`

This member informs the system whether the ability to utilize any software synthesizers installed in the host system is requested or allowed on a given `MIDIAccess` object. On `requestMIDIAccess()`, if this member is set to true, but software synthesizer support is denied (either by policy or by user action), the access request will fail with a "`NotAllowedError`" error. If this support is not requested, the system should not include any software synthesizers in the `MIDIAccess` exposure of available ports.

Note that may result in a two-step request procedure if software synthesizer support is desired but not required - software synthesizers may be disabled when MIDI hardware device access is allowed.

§ 5. The MIDI API

§ 5.1 ***MIDIInputMap*** Interface

```
WebIDL

[SecureContext, Exposed=(Window,Worker)] interface MIDIInputMap {
  readonly maplike <DOMString, MIDIInput>;
};
```

The `MIDIInputMap` is a maplike interface whose value is a `MIDIInput` instance and key is its ID.

This type is used to represent all the currently available MIDI input ports.

§ 5.2 ***MIDIOutputMap*** Interface

```
WebIDL

[SecureContext, Exposed=(Window,Worker)] interface MIDIOutputMap {
  readonly maplike <DOMString, MIDIOutput>;
};
```

The `MIDIOutputMap` is a maplike interface whose value is a `MIDIOutput` instance and key is its ID.

This type is used to represent all the currently available MIDI output ports.

§ 5.3 ***MIDIAccess*** Interface

This interface provides the methods to list MIDI input and output devices, and obtain access to an individual device.

```
WebIDL

[SecureContext, Exposed=(Window,Worker), Transferable] interface
MIDIAccess: EventTarget {
  readonly attribute MIDIInputMap inputs;
  readonly attribute MIDIOutputMap outputs;
  attribute EventHandler onstatechange;
  readonly attribute boolean sysexEnabled;
};
```

inputs

The MIDI input ports available to the system.

outputs

The MIDI output ports available to the system.

onstatechange

The handler called when a new port is connected or an existing port changes the state attribute.

This event handler, of type `MIDIConnectionEvent`, *MUST* be supported by all objects implementing the `MIDIAccess` interface.

> NOTE
>
> It is important to understand that leaving an `EventHandler` attached to this object will prevent it from being garbage-collected; when finished using the `MIDIAccess`, you should remove any `onstatechange` listeners.

Whenever a previously unavailable MIDI port becomes available for use, or an existing port changes the state attribute, the user agent *SHOULD* run the following steps:

1. Let *port* be the `MIDIPort` corresponding to the newly-available, or the existing port.
2. Fire an event named "statechange" at the `MIDIAccess`, using `MIDIConnectionEvent` with `port` set to *port*.

sysexEnabled

This attribute informs the user whether System Exclusive support is enabled on this MIDIAccess.

§ 5.4 ***MIDIPort*** Interface

This interface represents a MIDI input or output port.

WebIDL

```
[SecureContext, Exposed=(Window,Worker)] interface MIDIPort:
EventTarget {
  readonly attribute DOMString id;
```

```
    readonly attribute DOMString? manufacturer;
    readonly attribute DOMString? name;
    readonly attribute MIDIPortType type;
    readonly attribute DOMString? version;
    readonly attribute MIDIPortDeviceState state;
    readonly attribute MIDIPortConnectionState connection;
    attribute EventHandler onstatechange;
    Promise <MIDIPort> open();
    Promise <MIDIPort> close();
};
```

id

A unique ID of the port. This can be used by developers to remember ports the user has chosen for their application. The User Agent *MUST* ensure that the `id` is unique to only that port. The User Agent *SHOULD* ensure that the id is maintained across instances of the application - e.g., when the system is rebooted - and when a device is removed from the system. Applications may want to cache these ids locally to re-create a MIDI setup. Some systems may not support completely unique persistent identifiers; in such cases, it will be more challenging to maintain identifiers when another interface is added or removed from the system. (This might throw off the index of the requested port.) It is expected that the system will do the best it can to match a port across instances of the MIDI API: for example, an implementation may opaquely use some form of hash of the port interface manufacturer, name and index as the id, so that a reference to that port id is likely to match the port when plugged in. Applications may use the comparison of id of MIDIPorts to test for equality.

manufacturer

The manufacturer of the port.

name

The system name of the port.

type

A descriptor property to distinguish whether the port is an input or an output port. For `MIDIOutput`, this *MUST* be `"output"`. For `MIDIInput`, this *MUST* be `"input"`.

version

The version of the port.

state

The state of the device.

connection

The state of the connection to the device.

onstatechange

The handler called when an existing port changes its state or connection attributes.

This event handler, of type "statechange", *MUST* be supported by all objects implementing `MIDIPort` interface.

NOTE

It is important to understand that leaving an `EventHandler` attached to this object will prevent it from being garbage-collected; when finished using the `MIDIPort`, you should remove any `onstatechange` listeners.

open

Makes the MIDI device corresponding to the `MIDIPort` explicitly available. Note that this call is NOT required in order to use the `MIDIPort`- calling `send()` on a `MIDIOutput`, attaching a MIDIMessageEvent `EventHandler` on a `MIDIInput`, or adding a MIDIMessageEvent `EventListener` on a `MIDIInput` will cause an implicit open(). The underlying implementation may not need to do anything in response to this call. However, some underlying implementations may not be able to support shared access to MIDI devices, so using explicit open() and close() calls will enable MIDI applications to predictably control this exclusive access to devices.

When invoked, this method returns a Promise object representing a request for access to the given MIDI port on the user's system.

If the port device has a state of "connected", when access to the port has been obtained (and the port is ready for input or output), the vended Promise is resolved.

If access to a connected port is not available (for example, the port is already in use in an exclusive-access-only platform), the Promise is rejected (if any) is invoked.

If `open()` is called on a port that is "disconnected", the port's .connection will transition to "pending", until the port becomes "connected" or all references to it are dropped.

Multiple calls to `open()` are allowed even if not all vended Promises have settled.

When this method is called, the user agent *MUST* run the ***algorithm to open a MIDIPort***:

1. Let *promise* be a new Promise object and *resolver* be its associated resolver.
2. Return *promise* and run the following steps asynchronously.
3. Let *port* be the given `MIDIPort` object.

4. If the device's connection is already "open" (e.g. open() has already been called on this `MIDIPort`, or the port has been implicitly opened), jump to the step labeled *success* below.
5. If the device's connection is "pending" (i.e. the connection had been opened and the device was subsequently disconnected), jump to the step labeled *success* below.
6. If the device's state is "disconnected", change the `connection` attribute of the `MIDIPort` to "pending", and enqueue a new `MIDIConnectionEvent` to the statechange handler of the `MIDIAccess` and to the statechange handler of the `MIDIPort` and jump to the step labeled *success* below.
7. Attempt to obtain access to the given MIDI device in the system. If the device is unavailable (e.g. is already in use by another process and cannot be opened, or is disconnected), jump to the step labeled *failure* below. If the device is available and access is obtained, continue the following steps.
8. Change the `connection` attribute of the MIDIPort to `"open"`, and enqueue a new `MIDIConnectionEvent` to the statechange handler of the `MIDIAccess` and to the statechange handler of the `MIDIPort`.
9. If this port is an output port and has any enqueued send data with timestamps in the future, asynchronously begin sending that data.
10. ***success***: Call *resolver*'s `accept(value)` method with *port* as value argument.
11. Terminate these steps.
12. ***failure***: Let *error* be a new `DOMException`. This exception's .name should be `"InvalidAccessError"` if the port is unavailable.
13. Call *resolver*'s `reject(value)` method with *error* as value argument.

`close`

Makes the MIDI device corresponding to the `MIDIPort` explicitly unavailable (subsequently changing the state from "open" to "closed"). Note that successful invocation of this method will result in MIDI messages no longer being delivered to MIDIMessageEvent handlers on a `MIDIInput` (although setting a new handler will cause an implicit open()).

The underlying implementation may not need to do anything in response to this call. However, some underlying implementations may not be able to support shared access to MIDI devices, and the explicit close() call enables MIDI applications to ensure other applications can gain access to devices.

When invoked, this method returns a Promise object representing a request for access to the given MIDI port on the user's system. When the port has been closed (and therefore, in exclusive access systems, the port is available to other applications), the vended Promise is resolved. If the port is disconnected, the Promise is rejected.

Multiple calls to `close()` are allowed even if not all vended Promises have settled.

When the `close()` method is called, the user agent *MUST* run the following steps:

1. Let *promise* be a new Promise object and *resolver* be its associated resolver.
2. Return *promise* and run the following steps asynchronously.
3. Let *port* be the given `MIDIPort` object.
4. If the port is already closed (its .connection is "closed" - e.g. the port has not yet been implicitly or explicitly opened, or `close()` has already been called on this `MIDIPort`), jump to the step labeled ***closed*** below.
5. If the port is an input port, skip to the next step. If the output port's .state is not "connected" or if its .connection is "pending", clear all enqueued send data and skip to the next step. Clear any enqueued send data in the system with timestamps in the future, then finish sending any send messages with no timestamp or with a timestamp in the past or present, prior to proceeding to the next step.
6. Close access to the port in the underlying system if open, and release any blocking resources in the underlying system.
7. Change the `connection` attribute of the MIDIPort to `"closed"`, and enqueue a new `MIDIConnectionEvent` to the statechange handler of the `MIDIAccess` and to the statechange handler of the `MIDIPort`.
8. ***closed***: Call *resolver*'s `accept(value)` method with *port* as value argument.

Whenever the MIDI port corresponding to the `MIDIPort` changes the state attribute, the user agent *SHOULD* run the following steps:

1. Let *port* be the `MIDIPort`.
2. Fire an event named statechange at the `MIDIPort`, and statechange at the `MIDIAccess`, using `MIDIConnectionEvent` with the `port` attribute set to *port*.

§ 5.4.1 ***MIDIInput*** Interface

WebIDL

```
[SecureContext, Exposed=(Window,Worker)] interface MIDIInput: MIDIPort
{
  attribute EventHandler onmidimessage;
};
```

onmidimessage

This event handler, of type "midimessage", *MUST* be supported by all objects implementing `MIDIInput` interface.

If the handler is set and the state attribute is not `"opened"`, underlying implementation tries to make the port available, and change the state attribute to `"opened"`. If succeeded, `MIDIConnectionEvent` is delivered to the corresponding `MIDIPort` and `MIDIAccess`.

Whenever the MIDI port corresponding to the `MIDIInput` finishes receiving one or more MIDI messages, the user agent *MUST* run the following steps:

1. Let *port* be the `MIDIInput`.
2. If the `MIDIAccess` did not enable System Exclusive access, and the message is a System Exclusive message, abort this process.
3. Fire an event named "midimessage" at *port*, using `MIDIMessageEvent` with the `timeStamp` attribute set to the time the message was received by the system, and with the `data` attribute set to a `Uint8Array` of MIDI data bytes representing a single MIDI message.

It is specifically noted that MIDI System Real Time messages may actually occur in the middle of other messages in the input stream; in this case, the System Real Time messages will be dispatched as they occur, while the normal messages will be buffered until they are complete (and then dispatched).

§ 5.4.2 ***MIDIOutput*** Interface

WebIDL

```
[SecureContext, Exposed=(Window,Worker)] interface MIDIOutput :
MIDIPort {
  undefined send(sequence<octet> data, optional DOMHighResTimeStamp
timestamp = 0);
  undefined clear();
};
```

send

Enqueues the message to be sent to the corresponding MIDI port. The underlying implementation will (if necessary) coerce each member of the sequence to an unsigned 8-bit integer. The use of sequence rather than a Uint8Array enables developers to use the convenience of `output.send( [ 0x90, 0x45, 0x7f ] );` rather than having to create a Uint8Array, e.g. `output.send( new Uint8Array( [ 0x90, 0x45, 0x7f ] ) );`

The data contains one or more complete, valid MIDI messages. Running status is not allowed in the data, as underlying systems may not support it.

If *data* is not a valid sequence or does not contain a valid MIDI message, throw a `TypeError` exception.

If *data* is a System Exclusive message, and the `MIDIAccess` did not enable System Exclusive access, throw an `InvalidAccessError` exception.

If the port is "disconnected", throw an `InvalidStateError` exception.

If the port is "connected" but the connection is "closed", asynchronously try to open the port.

sequence<octet> data
The data to be enqueued, with each sequence entry representing a single byte of data.

optional DOMHighResTimeStamp timestamp
The time at which to begin sending the data to the port (as a `DOMHighResTimeStamp` - a number of milliseconds measured relative to the navigation start of the document). If `timestamp` is set to zero (or another time in the past), the data is to be sent as soon as possible. Multiple calls to `send()` with the same timestamp must result in the data being sent in the order the calls were made.

clear

Clears any enqueued send data that has not yet been sent from the `MIDIOutput`'s queue. The implementation will need to ensure the MIDI stream is left in a good state, so if the output port is in the middle of a sysex message, a sysex termination byte (0xf7) should be sent.

§ 5.4.3 *`MIDIPortType`* Enum

```
WebIDL
enum MIDIPortType {
  "input",
  "output",
};
```

`input`
: If a `MIDIPort` is an input port, the type member *MUST* be this value.

`output`
: If a `MIDIPort` is an output port, the type member *MUST* be this value.

§ 5.4.4 *`MIDIPortDeviceState`* Enum

```
WebIDL
enum MIDIPortDeviceState {
  "disconnected",
  "connected",
};
```

`disconnected`
: The device that `MIDIPort` represents is disconnected from the system. When a device is disconnected from the system, it should not appear in the relevant map of input and output ports.

`connected`
: The device that `MIDIPort` represents is connected, and should appear in the map of input and output ports.

§ 5.4.5 *`MIDIPortConnectionState`* Enum

```
WebIDL
```

```
enum MIDIPortConnectionState {
  "open",
  "closed",
  "pending",
};
```

open
: The device that `MIDIPort` represents has been opened (either implicitly or explicitly) and is available for use.

closed
: The device that `MIDIPort` represents has not been opened, or has been explicitly closed. Until a `MIDIPort` has been opened either explicitly (through `MIDIPort.open()`) or implicitly (by adding a midimessage event handler on an input port, or calling `MIDIOutput.send()` on an output port, this should be the default state of the device.

pending
: The device that `MIDIPort` represents has been opened (either implicitly or explicitly), but the device has subsequently been disconnected and is unavailable for use. If the device is reconnected, prior to sending a statechange event, the system should attempt to reopen the device (following the algorithm to open a MIDIPort); this will result in either the connection state transitioning to "open" or to "closed".

§ 5.5 ***MIDIMessageEvent*** Interface

An event object implementing this interface is passed to a `MIDIInput`'s onmidimessage handler when MIDI messages are received. Note that the DOM `Event` `timeStamp` attribute is defined as a `DOMHighResTimeStamp`, and represents the high-resolution time of when the event was received or is to be sent.

WebIDL
```
[SecureContext, Exposed=(Window,Worker)]
interface MIDIMessageEvent : Event {
  constructor(DOMString type, optional MIDIMessageEventInit
eventInitDict = {});
  readonly attribute Uint8Array? data;
};
```

data
: A Uint8Array containing the MIDI data bytes of a single MIDI message.

§ 5.5.1 *MIDIMessageEventInit* Dictionary

WebIDL
```
dictionary MIDIMessageEventInit: EventInit {
  Uint8Array data;
};
```

data
: A Uint8Array containing the MIDI data bytes of a single MIDI message.

§ 5.6 *MIDIConnectionEvent* Interface

An event object implementing this interface is passed to a `MIDIAccess`' `onstatechange` handler when a new port becomes available (for example, when a MIDI device is first plugged in to the computer), when a previously-available port becomes unavailable, or becomes available again (for example, when a MIDI interface is disconnected, then reconnected) and (if present) is also passed to the `onstatechange` handlers for any `MIDIPort`s referencing the port.

When a `MIDIPort` is in the "pending" state and the device is reconnected to the host system, prior to firing a statechange event the algorithm to open a MIDIPort is run on it to attempt to reopen the port. If this transition fails (e.g. the Port is reserved by something else in the underlying system, and therefore unavailable for use), the connection state moves to "closed", else it transitions back to "open". This is done prior to the statechange event for the device state change so that the event will reflect the final connection state as well as the device state.

Some underlying systems may not provide notification events for device connection status; such systems may have long time delays as they poll for new devices infrequently. As such, it is suggested that heavy reliance on connection events not be used.

WebIDL

```
[SecureContext, Exposed=(Window,Worker)]
interface MIDIConnectionEvent : Event {
  constructor(DOMString type, optional MIDIConnectionEventInit
eventInitDict = {});
  readonly attribute MIDIPort? port;
};
```

port

The port that has been connected or disconnected.

§ 5.6.1 *MIDIConnectionEventInit* Dictionary

WebIDL

```
dictionary MIDIConnectionEventInit: EventInit {
  MIDIPort port;
};
```

port

The port that has been connected or disconnected.

§ 6. Privacy Considerations

Allowing the enumeration of the user's MIDI interfaces is a potential target for fingerprinting; that is, uniquely identifying a user by the specific MIDI interfaces they have connected.

Note that in this context what can be enumerated is the MIDI *interfaces*. This includes most devices connected to the host computer with USB, since USB-MIDI devices typically have their own MIDI interface and would be enumerated. An individual sampler or synthesizer MIDI device plugged into a MIDI interface with a 5-pin DIN cable would not be enumerated. The interfaces that could be fingerprinted are equivalent to MIDI "ports", and for each MIDI interface the API will expose the name of the device, manufacturer, and opaque identifier of the MIDI interface.

Most systems have no MIDI interfaces attached. Few systems will have large numbers of MIDI interfaces attached. Thus, the additional fingerprinting exposure of enumerating MIDI devices is similar to the Gamepad API’s additional fingerprinting exposure through gamepad enumeration:

typical users will have at most a few devices connected, their configuration may change, and the information exposed is about the interface itself (i.e., no user-configured data).

§ 7. Security Considerations

The first MIDI devices were released in 1983, before the web platform and its security risks existed. Many MIDI devices are still in use long after their manufacturers stopped supporting them. MIDI has adapted to transports beyond the original serial connection, such as FireWire, USB, and Bluetooth. This poses a security challenge, with a long tail of devices from different eras that do not have official support but are still actively in use, connected to computers and the web in ways their designers did not expect.

§ 7.1 Malicious Firmware Updates

One concerning theoretical attack involves malicious firmware updates for USB-MIDI devices. USB devices in general can do things based on their device descriptor, which is sent from the USB device itself. If a USB-MIDI device's firmware can modify what descriptor is sent, it could make itself act as a human interface device. This could allow a malicious website to read or inject keystrokes or other events on the host computer, which could lead to a total compromise of the system.

The attack would proceed as follows:

1. A malicious site tricks the user into granting Web MIDI permission.
2. The malicious site enumerates the MIDI devices connected to the user's machine, and identifies a vulnerable device.
3. The malicious site sends a pre-crafted set of MIDI messages to the vulnerable device, compromising the device by overwriting its firmware and adding a human interface device to its USB descriptor.
4. The compromised device injects keystrokes to download or otherwise reproduce a pre-crafted security exploit and execute it, compromising the system.

In order to enable to the above attack, a MIDI device would need *all* of the following to be true:

- Be vulnerable to malicious firmware updates. *All* of the following must be true:

  - Have user-programmable firmware. Many MIDI devices have re-programmable firmware. Those that do not are not vulnerable.
  - Allow firmware updates via sending MIDI messages. Many MIDI devices use System Exclusive messages to perform firmware updates. This is a common use of System Exclusive messages. It is technically possible to make a device that uses non-System Exclusive messages to perform firmware updates, but this is not an intended use of non-System Exclusive messages. Other devices may use out-of-band USB communication or require the user to connect a storage device directly to the MIDI device to perform firmware updates, which cannot be done using the Web MIDI API, and are not vulnerable.
  - Allow firmware updates without explicit user interaction on the device. There are known MIDI devices which do not require any user interaction to initiate a firmware update. Most MIDI devices require the user to place them in a special update mode first, such as by holding down a button or selecting a menu option, and are not vulnerable unless the attacker tricks the user into cooperating.
- Be a USB-MIDI device. Most modern MIDI devices have a USB MIDI interface. Some older MIDI devices only have a serial connection, and would not enable this attack.
- The MIDI device's firmware would have to be able to modify the USB controller's firmware. Many MIDI devices have USB controller firmware that is not addressable from the main firmware, and would not enable this attack.

MIDI devices that are vulnerable to malicious firmware updates but do not satisfy the other conditions cannot be used with this attack to compromise the host system. A malicious firmware update could still cause these MIDI devices to stop working or behave in undesired ways.

To mitigate this risk, implementers should emphasize the following in their implementations:

- Implement `requestMIDIAccess()` in a way that informs users of the risks of firmware updates, such as through text in permission prompts.
- Implement SecureContext as specified to prevent user-initiated legitimate firmware updates from being modified.
- Handle the sysex parameter as specified, to encourage developers to only request System Exclusive messages when necessary, since most firmware updates use System Exclusive messages.

Explicitly allowing or blocking lists of known MIDI devices may also help mitigate this specific attack, but many small companies and individuals build MIDI devices, and many MIDI devices are no longer supported, so doing this would significantly reduce the usability of the Web MIDI API.

§ 7.2 Additional Security Considerations

Separate from the fingerprinting concerns of identifying the available ports are concerns around sending and receiving MIDI messages. Those issues are explored in more depth below.

MIDI messages can be divided into System Exclusive messages, and short (non-System Exclusive) messages. System Exclusive messages can be further subdivided into Universal System Exclusive messages such as the commonly recognized MIDI Time Code and MIDI Sample Dump Standard, and device-specific messages like “patch control data for a Roland Jupiter-80 synthesizer” that do not apply to other devices.

Before discussing security concerns, it's useful to examine what scenarios are enabled by MIDI using these features:

- Receiving short messages from MIDI devices - this enables getting input from keyboards, drum pads, guitars, wind controllers, DJ/controllerist controllers, and more, and using those messages as input to control instruments and features in the Web Audio API as well as other control scenarios. MIDI is the protocol of choice for the multi-billion-dollar music production industry for getting physical controllers like knobs and buttons attached to your computer, both in pro/prosumer audio and media applications as well as consumer applications like Garageband.
- Sending short messages to MIDI devices - it’s tempting to say sending is significantly less interesting, as the scenario of attached output devices like hardware synthesizers is less common in today's market. The major exception to this is that many MIDI controllers have external host control of their indicator lights, and this makes them dramatically more useful. For example, the very popular Novation Launchpad controller uses MIDI note on/off messages sent to it to turn on/off and change colors of the buttons. The same is true of nearly all DJ controllers.
- Sending and receiving System Exclusive messages - for more advanced communication with high-end hardware devices, System Exclusive messages are required. Some common MIDI commands are also sent as Universal System Exclusive messages, such as MIDI Machine Control - generic start/stop/rewind/fast-forward commands. Many devices use device-specific System Exclusive messages to program patches, send advanced controller messages, download firmware, etc., which are much-demanded scenarios for Web MIDI. Some devices use System Exclusive as a direct control protocol, as they can pack more data into a single “message”, and most devices use System Exclusive as a way to save and restore patches and configuration information on less-expensive computer storage. Several of the major music hardware producers have expressed strong interest in using Web MIDI to provide web-based configuration and programming interfaces to their hardware. In short, disabling System Exclusive altogether does not only disable high-end scenarios.

The potential security impact of each of these is as follows:

- Sending short messages to a MIDI device - sending note-on/note-off/controller messages could cause sounds to be played by attached devices, including (on Mac and Windows) any default virtual synthesizers. This by itself does not cause any concerning exposure - you can already make sounds without interaction, through <audio> or Web Audio. Some attached devices might be professional lighting control systems, so it’s possible to control stage lighting; however, this is rare, and no known system has the ability to cause lasting damage or information leakage based solely on short messages. At worst, a malicious page could flash lights, and the user could close the page and reset their lighting controller. The additional concerns about sending short messages are analogous to any audio output - you cannot overwrite user information or expose user information, but you can make sounds happen, change patches, or (in rare configurations) toggle lights - but non-destructively, and not persistently.
- Receiving short messages from a MIDI device - receiving note-on/note-off/controller messages would not cause information exposure or security issues, as there is no identifying data being received, just a stream of controller messages - all of which must be initiated by the user on that MIDI device (except clock-type messages). This is analogous to listening to keyboard, mouse, mobile/laptop accelerometer, touch input or gamepad events; there is no additional information exposed, and all messages other than clock signals must be initiated by the user.
- Sending and receiving System Exclusive messages - this is the biggest concern, because it is possible to write code that looks for system-specific responses to System Exclusive messages, which could identify the hardware available, and then use it to download data - e.g. samples stored in a sampler - or replace that data (erasing sample data or patches in the device), although both these scenarios would have to be coded for a particular device. It is also possible that some samplers might enable a System Exclusive message to start recording a sample - so if the sampler happened to have a dedicated microphone attached (uncommon in practice, but possible), it would be possible to write code specific to a particular device that could record a short sample of sound and then upload it to the network without further user intervention. You could not stream audio from the device, and most samplers have fairly limited memory, and MIDI Sample Dump sysex is a slow way to transfer data - it has to transcode into 7-bit - so it’s unlikely you could listen in for long periods. More explicit fingerprinting is a concern, as the patch information/stored samples/user configuration could uniquely identify the system. Again, this requires much device-specific code; there is not standardized “grab all patches and hash it” capability. This suggests that System Exclusive messages are in a security category of their own. Because of this less bounded potential, it seems that distinguishing requests for SysEx separately in the API is a good idea, in order to more carefully provide user security hooks. The suggested security model explicitly allows user agents to require the user's approval before giving access to

MIDI devices, although it is not currently required to prompt the user for this approval - but it is also detailed that System Exclusive support must be requested as part of that request.

§ 8. Changelog

§ 8.1 Changes since Working Draft 17 March 2015

- Merge pull request #272 from mjwilson-google/examples
- Update text
- Move remaining examples to explainer, and modernize and fix errors in…
- Merge pull request #271 from mjwilson-google/security
- Address reviewer comments
- Merge branch 'WebAudio:gh-pages' into security
- Remove spaces around slashes
- Add more detail to security section
- Merge pull request #270 from mjwilson-google/var
- Use let instead of var in examples
- Merge pull request #269 from mjwilson-google/valid-midi-message
- Revise wording for normative definition
- Add non-normative definition of a valid MIDI message
- Merge pull request #267 from mjwilson-google/not-allowed-error
- Merge pull request #268 from mjwilson-google/privacy-considerations
- Change the majority of to most
- Update privacy section
- Use NotAllowedError instead of SecurityError
- Merge pull request #265 from mjwilson-google/unlink-definitions
- Merge pull request #264 from mjwilson-google/clarify-clear
- Don't use <dfn> tag on algorithms that aren't referenced elsewhere in…

- Update README.md to reference the gh-pages branch
- Clarify that MIDIOutput.clear() should clear enqueued send data
- Merge pull request #263 from mjwilson-google/update-token
- Update echidna token name
- Merge pull request #256 from mjwilson-google/worker-availability
- Merge pull request #262 from mjwilson-google/spec-prod
- Remove extra parameter from p tag
- Add auto-publish.yml
- Merge pull request #259 from mjwilson-google/changelog
- Add changelog
- Expose all interfaces
- Make MIDIAccess transferable and exposed to workers
- Merge pull request #255 from mjwilson-google/link-definitions
- Add links and update link format according to Respec rules
- Merge pull request #252 from mjwilson-google/168-require-init
- Don't need to explicitly nullify dictionaries
- Make types nullable
- Merge branch 'WebAudio:gh-pages' into 168-require-init
- Merge pull request #254 from mjwilson-google/link-webaudio
- Link Web Audio API to definition and add some punctuation
- Merge pull request #253 from mjwilson-google/242-no-implementation-re…
- Add statement that no implementation report currently exists
- Merge pull request #251 from mjwilson-google/add-test-suite-uri
- Make init dictionaries required
- Add test suite URI
- Merge pull request #250 from mjwilson-google/185-split-privacy-and-se…
- Split the existing Security and Privacy Considerations section into t…
- Merge pull request #249 from mjwilson-google/fix-headings
- Organize sections

- Merge pull request #248 from mjwilson-google/174-move-examples-to-exp…
- Move examples to explainer
- Merge pull request #247 from mjwilson-google/244-system-real-time
- Merge branch 'gh-pages' into 244-system-real-time
- Merge pull request #246 from mjwilson-google/244-system-exclusive
- Merge branch 'WebAudio:gh-pages' into 244-system-exclusive
- Wrap at 80 columns
- Link definition of System Real Time
- Capitalize and link System Exclusive definition
- Merge pull request #245 from mjwilson-google/default-allowlist
- Fix links to user agent and default allowlist
- Merge pull request #243 from mjwilson-google/move-sotd
- Move sotd to just after abstract
- Merge pull request #241 from mjwilson-google/editor-maintenance
- Move Jussi to former editor, add URL for Michael
- Merge pull request #237 from WebAudio/mp-edit
- Update index.html
- fix for #142 https://github.com/WebAudio/web-midi-api/issues/142
- Merge pull request #234 from dontcallmedom/respec-fixes
- Merge pull request #224 from marcoscaceres/dont_exclude
- Merge pull request #236 from WebAudio/mp-edit
- fix for issue #205 https://github.com/WebAudio/web-midi-api/issues/205
- Simplify pharsing of event firing
- Use group's shortname as now required by respec
- Editorial: don't exclude Permissions IDL
- Merge pull request #222 from tidoust/flag-midipermissiondescriptor
- Flag MidiPermissionDescriptor definition as non normative
- Merge pull request #221 from marcoscaceres/patch-1
- Merge pull request #219 from miketaylr/permissions/issue/294

- Editorial: fix event.timeStamp typo
- Fold in @marcoscaceres' suggestions.
- Chore: Move specs from data-cite to xref config.
- Editorial: Add a Permissions integration section
- Editorial: Edits to permissions policy section.
- Chore: stray whitespace cleanup
- Editorial: Default allowlist value should be a string, not a sequence.
- Merge pull request #218 from ZingBlue/patch-1
- Use HTTPS in URLs
- Update URL to use github.io
- Merge pull request #214 from autokagami/webmidi
- Merge pull request #213 from sidvishnoi/gh-pages
- Merge pull request #209 from hughrawlinson/patch-1
- Editorial: Align with Web IDL specification
- Editorial: rename feature-policy to permissions-policy
- chore: fix feature-policy linking errors
- Remove errant character
- chore: tidy doc
- Editorial: use new constructor syntax
- Chore: ReSpec fixes + enable xref
- Additional fixes
- fix Exposed warnings
- Chore: ReSpec fixes + enable xref
- chore(.pr-preview.json): enable pull request previews
- set default for optional dictionary
- Chore(.pr-preview.json): enable pull request previews
- set default for optional dictionary
- Merge pull request #200 from toyoshim/securecontext_maplikes
- Making MIDI{Input|Output}Map interfaces as [SecureContext]

- Merge pull request #196 from WebAudio/dontcallmedom-patch-1
- Mark repo as host of rec-track work
- Merge pull request #194 from WebAudio/fixup
- fixup IDL, cross refs
- Merge pull request #192 from cwilso/gh-pages
- Resolve review comments
- Marking interfaces as [SecureContext]
- Merge pull request #188 from foolip/update-wpt-url
- Update web-platform-tests URLs
- Adding baseline CODE_OF_CONDUCT.md
- Create CONTRIBUTING.md
- Create LICENSE.md
- Create w3c.json
- Merge pull request #181 from raymeskhoury/gh-pages
- Merge pull request #182 from foolip/patch-1
- Describe that a SecurityError should be returned if the request is de…
- Ask for tests for normative changes in README.md
- Add Feature Policy integration for WebMidi
- Merge pull request #178 from eyqs/gh-pages
- Silence respec whining about https
- Fix minor typos
- Merge pull request #175 from WebAudio/modernize_idl
- chore: use new school WebIDL (closes #170)
- Clarify onstatechange's effect on lifetime.
- Add a "software" option to enable requests of software synths.
- Remove receivedTime (in favor of DOM Event timeStamp)
- More permissive copyright license
- WD->ED
- Merge pull request #130 from natevw/patch-1

- Fix typo (fixes issue #163)
- Merge pull request #155 from natevw/patch-2
- Fix up inputs enumeration example
- Remove callback references
- Fix up double->DOMHighResTimeStamp references.
- default value for timestamp
- Removed reference to software synthesizers. Fixes issue #153.
- Fixes #149.
- Merge pull request #147 from ryoyakawai/submission/update
- Fixed #146 : Removed "value." between "entry." and "onmidimessage" in…
- Remove reference to Typed-Arrays for Uint8Array
- https for editor's draft link
- REALLY fix #135.
- Typo correction: fixes #135.
- "state"->"connection"
- Fixes #132.
- Fixes #131.
- fix typo in close method
- Merge pull request #275 from mjwilson-google/event-listener
- Specify adding an EventListener also opens MIDIInput ports
- Merge pull request #273 from mjwilson-google/send-timing
- Merge pull request #274 from mjwilson-google/pending-close

§ A. References

§ A.1 Normative references

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

[hr-time]
: *High Resolution Time*. Yoav Weiss. W3C. 7 November 2024. W3C Working Draft. URL: https://www.w3.org/TR/hr-time-3/

[html]
: *HTML Standard*. Anne van Kesteren; Domenic Denicola; Dominic Farolino; 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/

[MIDI]
: *MIDI 1.0 Core Specifications*. The MIDI Manufacturers Association. 2014. URL: https://midi.org/midi-1-0-core-specifications

[Permissions]
: *Permissions*. Marcos Caceres; Mike Taylor. W3C. 20 December 2024. W3C Working Draft. URL: https://www.w3.org/TR/permissions/

[permissions-policy]
: *Permissions Policy*. Ian Clelland. W3C. 13 January 2025. W3C Working Draft. URL: https://www.w3.org/TR/permissions-policy-1/

[RFC2119]
: *Key words for use in RFCs to Indicate Requirement Levels*. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc2119

[RFC8174]
: *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc8174

[webaudio]
Web Audio API. Paul Adenot; Hongchan Choi. W3C. 17 June 2021. W3C Recommendation. URL: https://www.w3.org/TR/webaudio-1.0/

[WEBIDL]
Web IDL Standard. Edgar Chen; Timothy Gu. WHATWG. Living Standard. URL: https://webidl.spec.whatwg.org/