MIDI 1.0 Runtime (MidiManager)

This page documents the MIDI 1.0 runtime API centered around `jp.kshoji.unity.midi.MidiManager` and the `IMidiPlugin` backend interface.

> Note: The project supports multiple transports (native platform MIDI, WebGL MIDI, RTP-MIDI, Nearby Connections, etc.). They all present the same surface API through `IMidiPlugin` and `MidiManager`.

For install/build requirements, see:

- Getting Started
- Build PostProcessing & Scripting Define Symbols
- Platforms & Limitations

Core responsibilities

`MidiManager` is responsible for:

- Initializing/terminating the active MIDI plugin backend.
- Receiving MIDI 1.0 messages (Note On/Off, CC, PC, aftertouch, pitch wheel, SysEx, system messages).
- Dispatching messages to Unity objects via **Unity EventSystem** handler interfaces.
- Sending MIDI 1.0 messages through the current backend.

Initialization / Termination (recommended pattern)

Initialize in `Awake`, terminate in `OnDestroy`:

```
using UnityEngine;
using jp.kshoji.unity.midi;

public sealed class Midi1LifecycleExample : MonoBehaviour
{
    private void Awake()
    {
        MidiManager.Instance.RegisterEventHandleObject(gameObject);
        MidiManager.Instance.InitializeMidi(() =>
        {
#if (UNITY_ANDROID || UNITY_IOS || UNITY_WEBGL) && !UNITY_EDITOR
            // BLE MIDI only (where supported): start scan after init completes.
            MidiManager.Instance.StartScanBluetoothMidiDevices(0);
#endif
        });
    }

    private void OnDestroy()
    {
        MidiManager.Instance.TerminateMidi();
    }
}
```

> If you already have an `EventSystem` in your scene and see duplication warnings/errors, remove the `EventSystem` auto-add inside the manager's initialization.

Backend abstraction: IMidiPlugin

`IMidiPlugin` defines the platform/transport-dependent implementation:

- Lifecycle:
  - `InitializeMidi(Action initializeCompletedAction)`
  - `TerminateMidi()`
  - (Editor) `PlayModeStateChanged(...)`
- Optional BLE scan/advertise (platform gated by compile symbols).
- Device metadata:
  - `GetDeviceName(deviceId)`
  - `GetVendorId(deviceId)`
  - `GetProductId(deviceId)`
- MIDI 1.0 sending methods:
  - Note On/Off, CC, PC, aftertouch, pitch wheel
  - SysEx and system real-time/common messages (availability differs by platform)

Concrete implementations live in:

- `MidiPlugin.Android.cs`, `MidiPlugin.Apple.cs`, `MidiPlugin.Linux.cs`, `MidiPlugin.Windows.cs`, `MidiPlugin.WebGL.cs`
- `MidiPlugin.RtpMidi.cs` (RTP-MIDI transport)
- `MidiPlugin.Nearby.cs` (Nearby transport)

Device metadata: name / vendorId / productId

- `MidiManager.Instance.GetDeviceName(deviceId)`
- `MidiManager.Instance.GetVendorId(deviceId)`
- `MidiManager.Instance.GetProductId(deviceId)`

Important notes (mirrors the manual):

- Some platforms/transports do not support vendor/product IDs; empty strings may be returned.
- After a device is disconnected, these methods may return empty strings.

Availability of GetVendorId / GetProductId

Platform	**Bluetooth MIDI**	**USB MIDI**	**Network MIDI (RTP-MIDI)**	**Nearby Connections MIDI**
iOS	○	○	-	-
Android	○	○	-	-
Universal Windows Platform	-	○	-	-
Standalone macOS / Unity Editor macOS	○	○	-	-
Standalone Linux / Unity Editor Linux	-	-	-	-
Standalone Windows / Unity Editor Windows	-	○	-	-
WebGL	-	△ (GetVendorId only)	-	-

VendorId examples

Since the underlying platform API differs, the observed `VendorId` string can differ across OSes.

Platform	**Bluetooth MIDI**	**USB MIDI**	**Network MIDI (RTP-MIDI)**	**Nearby Connections MIDI**
iOS	QUICCO SOUND Corp.	Generic	-	-
Android	QUICCO SOUND Corp.	1410	-	-
Universal Windows Platform	-	VID_0582	-	-

Platform	Bluetooth MIDI	USB MIDI	Network MIDI (RTP-MIDI)	Nearby Connections MIDI
Standalone macOS / Unity Editor macOS	QUICCO SOUND Corp.	Generic	-	-
Standalone Linux / Unity Editor Linux	-	-	-	-
Standalone Windows / Unity Editor Windows	-	1	-	-
WebGL	QUICCO SOUND Corp.	Microsoft Corporation	-	-

ProductId examples

Same story for `ProductId`.

Platform	Bluetooth MIDI	USB MIDI	Network MIDI (RTP-MIDI)	Nearby Connections MIDI
iOS	mi.1	USB2.0-MIDI	-	-
Android	mi.1	298	-	-
Universal Windows Platform	-	PID_012A	-	-
Standalone macOS / Unity Editor macOS	mi.1	USB2.0-MIDI	-	-
Standalone Linux / Unity Editor Linux	-	-	-	-
Standalone Windows / Unity Editor Windows	-	102	-	-
WebGL	mi.1	UM-ONE	-	-

Event dispatch model (Unity EventSystem)

Handler interfaces

MIDI 1.0 event handler interfaces are defined in:

- `Assets/MIDI/Scripts/IMidiEventHandler.cs`
- `Assets/MIDI/Scripts/IMidiDeviceEventHandler.cs`

They are small, “one-event” interfaces such as:

- `IMidiNoteOnEventHandler`
- `IMidiControlChangeEventHandler`
- `IMidiSystemExclusiveEventHandler`
- … plus system message handlers for Start/Stop/Clock, Song Select, etc.

There are also aggregate interfaces:

- `IMidiPlayingEventsHandler` (note + channel events)
- `IMidiSystemEventsHandler` (system and SysEx)
- `IMidiAllEventsHandler` (everything)

Registering handlers

`MidiManager` exposes methods to register/unregister event handler objects. The expectation is:

- You implement one or more `IMidi*EventHandler` interfaces on a Unity component/object.
- You register that object with `MidiManager`.
- On incoming MIDI messages, `MidiManager` will execute callbacks for all matching handler interfaces.

Receiving events via C# event delegates (optional)

As an alternative to interfaces, you can subscribe to C# events named like:

- `MidiManager.Instance.OnMidiXXXXEvent`

Example:

```
using UnityEngine;
using jp.kshoji.unity.midi;

public sealed class Midi1DelegateExample : MonoBehaviour
{
    private void OnEnable()
    {
        MidiManager.Instance.OnMidiNoteOnEvent += OnMidiNoteOn;
    }

    private void OnDisable()
    {
        MidiManager.Instance.OnMidiNoteOnEvent -= OnMidiNoteOn;
    }

    private void OnMidiNoteOn(string deviceId, int group, int channel, int note,
int velocity)
        => Debug.Log($"NoteOn dev:{deviceId} ch:{channel} note:{note} vel:
{velocity}");
}
```

Bluetooth MIDI Peripheral mode (Android only)

In addition to scanning/connecting to BLE MIDI devices, Android can advertise your app as a **BLE MIDI Peripheral**.

API surface:

- Start advertising: `MidiManager.Instance.StartAdvertisingBluetoothMidiDevice()`
- Stop advertising: `MidiManager.Instance.StopAdvertisingBluetoothMidiDevice()`

Notes:

- This feature is **Android-only**.
- Advertising is separate from scanning; you can still use normal MIDI send/receive once a peer connects.
- If you are also using advanced Android BLE pairing workflows, see `FEATURE_ANDROID_COMPANION_DEVICE` in: Build PostProcessing & Scripting Define Symbols

Sending MIDI 1.0 messages

`MidiManager` offers helper methods for common messages:

- Note On/Off: `SendMidiNoteOn`, `SendMidiNoteOff`
- CC: `SendMidiControlChange`
- Program Change: `SendMidiProgramChange`
- Aftertouch: `SendMidiPolyphonicAftertouch`, `SendMidiChannelAftertouch`
- Pitch Wheel: `SendMidiPitchWheel`
- SysEx: `SendMidiSystemExclusive`
- System messages: clock, start/continue/stop, reset, etc.

Parameter ranges (common)

- `group`: 0–15 (often ignored for MIDI 1.0 backends, but included for API consistency)
- `channel`: 0–15
- `note`: 0–127
- `velocity`: 0–127
- `pressure`: 0–127
- `amount` (pitch wheel): 0–16383
- SysEx: byte array typically starting with `0xF0` and ending with `0xF7`

RTP-MIDI helper methods (if enabled)

When the RTP-MIDI plugin is used, `MidiManager` exposes methods to:

- Start/stop RTP-MIDI servers per port
- Connect/disconnect endpoints
- Query whether a listener is running

See also:

- Transports & Platform Notes

Source code example implementation (MIDI 1.0 quick start)

Create a new script (or use the provided one):

- `Assets/MIDI/Samples/Scripts/DocumentationExamples/Midi1QuickStartExample.cs`

Attach it to a GameObject, enter Play Mode, then connect a MIDI device.
It will:

- log device attach/detach
- log incoming Note On
- optionally send a test note to the first output device