MIDI 2.0 / UMP Runtime (Midi2Manager)

This page documents the MIDI 2.0 runtime, Universal MIDI Packet (UMP) parsing, and UDP MIDI 2.0 features as exposed by `jp.kshoji.unity.midi.Midi2Manager`.

For platform availability and OS constraints, see:

- Platforms & Limitations

Core responsibilities

`Midi2Manager` is responsible for:

- Initializing/terminating MIDI 2.0 plugin backends (depending on platform).
- Receiving **raw UMP words** (`uint[]`) and:
  - optionally forwarding “raw UMP” callbacks,
  - decoding message types into strongly typed events,
  - reassembling multi-part messages (e.g., SysEx8/text) where applicable.
- Sending UMP messages through the active backend.
- Hosting and connecting to **UDP MIDI 2.0** endpoints (server/client).
- Discovering UDP MIDI 2.0 servers (LAN discovery).

Initialization / Termination (recommended pattern)

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

public sealed class Midi2LifecycleExample : MonoBehaviour
{
    private void Awake()
    {
        MidiManager.Instance.RegisterEventHandleObject(gameObject);

        // MIDI 2.0 init
        MidiManager.Instance.InitializeMidi2(() => { });
    }

    private void OnDestroy()
    {
        MidiManager.Instance.TerminateMidi2();
    }
}
```

> This plugin routes MIDI 2.0 and MIDI 1.0 compatibility events through the same manager instance, so you will see both “MIDI 2” and “MIDI 1 compatibility” handler families in the API surface.

Backend abstraction

MIDI 2.0 plugin implementations derive from:

- `Midi2PluginBase`

Implementations include:

- `Midi2Plugin.Android.cs`
- `Midi2Plugin.Apple.cs`
- `Midi2Plugin.Linux.cs`
- `Midi2Plugin.Udp.cs` (network transport)

UMP event handler interfaces

UMP/MIDI 2.0 event handler interfaces are defined in:

- `Assets/MIDI/Scripts/IMidi2EventHandler.cs`
- `Assets/MIDI/Scripts/IMidi2DeviceEventHandler.cs`

Interfaces are grouped by UMP message type:

- **Utility (type 0)**
- **System Common & Realtime (type 1)**
- **MIDI 1.0 Channel Voice (type 2)** (compatibility)
- **Data 64-bit / SysEx (type 3)**
- **MIDI 2.0 Channel Voice (type 4)**
- **Data 128-bit / SysEx8 / Mixed Data Set (type 5)**
- **Flex Data (type D)**
- **UMP Stream (type F)**

Aggregate interfaces include:

- `IMidi2AllEventsHandler`
- `IUmpAllEventsHandler` (covers decoded MIDI 1.0 + MIDI 2.0 events from UMP)

Receiving events via C# event delegates (optional)

In addition to implementing handler interfaces, you can subscribe to manager events (delegate-based callbacks).

These are named like:

- `MidiManager.Instance.OnMidi2XXXXEvent`
- and (for UMP-encoded MIDI 1 compatibility) `MidiManager.Instance.OnMidi1XXXXEvent`

Example:

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

public sealed class Midi2DelegateExample : MonoBehaviour
{
    private void OnEnable()
    {
        MidiManager.Instance.OnMidi2OutputDeviceAttachedEvent +=
OnMidi2OutputDeviceAttached;
        MidiManager.Instance.OnMidi2NoteOnEvent += OnMidi2NoteOn;
    }

    private void OnDisable()
    {
        MidiManager.Instance.OnMidi2OutputDeviceAttachedEvent -=
OnMidi2OutputDeviceAttached;
        MidiManager.Instance.OnMidi2NoteOnEvent -= OnMidi2NoteOn;
    }

    private void OnMidi2OutputDeviceAttached(string deviceId)
        => Debug.Log($"MIDI 2.0 output attached: {deviceId}");

    private void OnMidi2NoteOn(string deviceId, int group, int channel, int note,
int velocity, int attributeType, int attributeData)
        => Debug.Log($"MIDI2 NoteOn dev:{deviceId} g:{group} ch:{channel} note:
{note} vel:{velocity}");
}
```

Sending MIDI 2.0 vs MIDI 1 compatibility messages

You can send either:

- MIDI 2.0 messages (higher precision, e.g. 16-bit velocity)
- MIDI 1.0 compatibility messages encoded in UMP (7/14-bit style)

Example (mirrors the original manual):

```
// MIDI 2.0 Note On (16-bit velocity)
MidiManager.Instance.SendMidi2NoteOn(
    "deviceId",
    0 /*group*/,
    0 /*channel*/,
    60 /*note*/,
    65535 /*velocity*/,
    0 /*attributeType*/,
    0 /*attributeData*/
);

// MIDI 1.0 compatibility Note On (7-bit velocity) encoded as UMP
MidiManager.Instance.SendMidi1NoteOn(
    "deviceId",
    0 /*group*/,
    0 /*channel*/,
    60 /*note*/,
    127 /*velocity*/
);
```

UDP MIDI 2.0 features

The UDP MIDI 2.0 transport (Network MIDI 2.0) is implemented by `Midi2Plugin.Udp.cs` and exposed via `MidiManager` / `Midi2Manager`.

Start the UDP MIDI 2.0 server

You can start a server with:

- endpoint name
- mDNS service instance name
- UDP port
- optional NAK/error callback
- optional authentication (shared secret or user auth list)

Example (placeholders for any secret/password values):

```
#if !UNITY_WEBGL || UNITY_EDITOR
// Start a UDP MIDI 2.0 server
MidiManager.Instance.StartUdpMidi2Server(
    12345,
    "MyMidi2Endpoint",
    "MyMIDI2Service"
);

// With NAK handling
MidiManager.Instance.StartUdpMidi2Server(
    12345,
    "MyMidi2Endpoint",
    "MyMIDI2Service",
    onErrorAction: nak =>
    {
        Debug.LogError($"UDP MIDI error. status:{nak.NakStatus}, command:
{nak.CommandHeader:x8}, message:{nak.Message}");
    }
);

// With shared secret authentication (placeholder)
MidiManager.Instance.StartUdpMidi2Server(
    12345,
    "MyMidi2Endpoint",
    "MyMIDI2Service",
    authenticationSecret: "<shared-secret>"
);

// With user authentication (placeholders)
MidiManager.Instance.StartUdpMidi2Server(
    12345,
    "MyMidi2Endpoint",
    "MyMIDI2Service",
    userAuthentications: new
```

```
System.Collections.Generic.List<System.Collections.Generic.KeyValuePair<string,
string>>
    {
        new("<account-1>", "<password-1>"),
        new("<account-2>", "<password-2>"),
    }
);
#endif
```

Discover and connect

```
#if !UNITY_WEBGL || UNITY_EDITOR
// Start discovery
MidiManager.Instance.StartDiscoverUdpMidi2Server();

// Read discovered services (service instance names)
var discovered = MidiManager.Instance.GetDiscoveredUdpMidi2Servers();

// Connect to the first discovered server
if (discovered.Count > 0)
{
    MidiManager.Instance.ConnectToUdpMidi2Server(discovered[0], "UDP-MIDI2-
Client");
}
#endif
```

Limitations:

- UDP MIDI 2.0 is still considered experimental; interoperability with other stacks may vary.

UmpSequencer (MIDI 2.0 clip sequencing)

The project includes a MIDI 2.0 sequencer workflow (separate from SMF sequencing):

- Reads/writes MIDI 2.0 Clip files (`.midi2`)
- Can record UMP events into a sequence model
- Can play a recorded clip/sequence
- Can convert between SMF (MIDI 1.0 `Sequence`) and UMP clip sequences

See also:

- SMF / Sequencing (SMF + conversion notes)

Clip/Container format notes

- MIDI 2.0 Clip file support exists, but Clip files are still not widespread.
- MIDI 2.0 Container file support exists, but the Container specification is draft-state and may change.
    - Consider the Container workflow experimental.

Source code example implementation (MIDI 2.0 quick start)

Use:

- `Assets/MIDI/Samples/Scripts/DocumentationExamples/Midi2QuickStartExample.cs`

Attach it to a GameObject and enter Play Mode.
It will:

- initialize MIDI 2.0 (`InitializeMidi2`)
- log MIDI 2.0 device attach/detach
- log MIDI 2.0 Note On events