

# SMF / Sequencing (jp.kshoji.midisystem)

---

This page documents the SMF (Standard MIDI File) and sequencing-related code under:

- [Assets/MIDI/Scripts/midisystem/](#)

Namespace: [jp.kshoji.midisystem](#)

## What this module provides

- An in-memory model of a MIDI sequence:
  - `Sequence`
  - `Track`
  - `MidiEvent`
- MIDI message representations:
  - `MidiMessage` (base)
  - `ShortMessage` (most channel/system messages)
  - `SysexMessage`
  - `MetaMessage`
- SMF file I/O:
  - `StandardMidiFileReader`
  - `StandardMidiFileWriter`
- Sequencing interfaces and implementation:
  - `ISequencer`, `SequencerImpl`
  - `IReceiver`, `ITransmitter`
  - Meta/controller event listeners

This is conceptually similar to the Java MIDI API model (Sequence/Track/MidiEvent).

## StandardMidiFileWriter behavior (important details)

When writing a `Sequence`:

- Supports SMF type 0 and type 1 (depending on track count).
- Writes `MThd` header with:
  - file type
  - number of tracks
  - division type/resolution encoding
- For each track:
  - Computes track length before writing.
  - Skips **system realtime** messages (`ShortMessage` status  $\geq$  `0xF8`) when writing.
  - Writes:
    - delta time (variable-length int)
    - message bytes (`ShortMessage`), or
    - `0xFF` + meta type + length + data (`MetaMessage`), or
    - `0xF0` + length + data (`SysexMessage`)
  - Ensures an **End of Track** meta event exists; writes one if missing.

# Sequencer workflows (playback / recording / export)

This section mirrors the shipped manual's sequencer usage patterns.

## Creating and opening a sequencer

```
using jp.kshoji.midisystem;

var isSequencerOpened = false;

// Create a sequencer and open it
var sequencer = new SequencerImpl(() => { isSequencerOpened = true; });
sequencer.Open();
```

## Read SMF into a Sequence and play it

```
using System.IO;
using jp.kshoji.midisystem;

sequencer.UpdateDeviceConnections();

using var stream = new FileStream(smfPath, FileMode.Open, FileAccess.Read);
sequencer.SetSequence(stream);

sequencer.Start();

// ... Later
sequencer.Stop();
```

## Record a sequence

```
using jp.kshoji.midisystem;

sequencer.UpdateDeviceConnections();

// PPQ example mirrors the manual
sequencer.SetSequence(new Sequence(Sequence.Ppq, 480));

sequencer.StartRecording();

// ... Later
sequencer.Stop();
```

## Write the recorded sequence to an SMF file

```
using System.IO;
using jp.kshoji.midisystem;

var sequence = sequencer.GetSequence();

if (sequence.GetTickLength() > 0)
{
    using var stream = new FileStream(recordedSmfPath, FileMode.Create,
FileAccess.Write);
    MidiSystem.WriteSequence(sequence, stream);
}
```

## Relationship to UMP tools

The UMP tooling in `Assets/MIDI/Scripts/UmpSequencer/` provides conversion between:

- SMF `Sequence` (MIDI 1.0 representation)
- UMP `UmpSequence` (MIDI 2.0 packet-oriented representation)

Use this if you need to:

- import/export MIDI files (SMF)
- convert to UMP clips for MIDI 2.0 workflows
- convert MIDI 2.0 clip files to SMF (or vice versa), where supported by the converter.

## Practical guidance

- For simple “play a MIDI file by sending messages out”, you typically:
  1. Read SMF into a `Sequence`
  2. Iterate events by tick (or use a sequencer)
  3. Send via `MidiManager` (MIDI 1.0) or convert to UMP and send via `Midi2Manager` (MIDI 2.0)
- If you need precise timing and transport controls (play/stop/loop), consider using:
  - `SequencerImpl` (SMF-style sequencing), or
  - the MIDI 2.0 UMP clip sequencer workflow if your target is UMP-centric (see [MIDI 2.0 / UMP](#)).

## Source code example implementation (SMF → MIDI out)

Use:

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

How to test:

1. Put a `.mid` file in `Assets/StreamingAssets/` (create the folder if needed)
2. Set `streamingAssetsMidiFileName` (e.g. `test.mid`)
3. Attach the component to a GameObject and enter Play Mode

This example reads the file via `StandardMidiFileReader`, iterates events, and sends basic messages via `MidiManager`.