Sample Code

Incorporating MIDI 2 into your apps

Add precision and improve musical control for your MIDI apps.

Download

iOS 14.0+ | iPadOS 14.0+ | Mac Catalyst 14.0+ | Xcode 16.2+

Overview

The MIDI 2 standard defines the Universal MIDI Packet (UMP) structure for the MIDI protocol transfer. Both MIDI 1 and 2 use the same structure. The framework expresses a MIDI 1 packet as MIDI-1UP, and the group field in UMP allows you to send up to 16 combined MIDI streams at the same time.

The MIDI 1 specification includes only the MIDI 1 protocol. The MIDI 2 specification includes the protocols MIDI 1, MIDI-1UP, MIDI 2, and the MIDI Capability Inquiry (MIDI-CI). The only requirement for you to provide a MIDI 2 device is bidirectional communication, which is necessary for MIDI-CI.

This sample code project shows how to use Core MIDI in MIDI Services. The sample consists of two targets built in Mac Catalyst that you use to send or receive MIDI events. Use the targets on the same device to show the transmission of UMP MIDI data.

Configure the sample code project

To run this sample app in macOS or iPadOS:

1. Build and launch the UMP Receiver target in Xcode.
2. In the UMP Receiver app, create a MIDI destination by providing a name for the destination and clicking or tapping the Create Destination button.
3. If the app is running on iPad, switch the receiver app to a Slide Over window.

4. Build and launch the UMP Send target in Xcode.

5. In the UMP Send app, select a MIDI message type to send, select a destination, and click or tap the Send button.

Understand MIDI-1UP and MIDI 2

By default, Core MIDI adopts the MIDI 2 specification and transparently converts sent MIDI data into the packet format and protocol a MIDI destination specifies. Regardless of the source packet format and protocol, legacy MIDI destinations receive `MIDIPacketList`. MIDI 2 destinations receive `MIDIEventList`.

Build a Universal MIDI Packet

The UMP send target shows how to build and send UMP events to a MIDI destination a person selects. The user interface allows for selecting the type of MIDI message to send before customizing the different elements of the packet. The visualizer provides a tabular view of the contents of the packet in decimal, binary, and hexadecimal form.

The app stores the packet state in `PacketModel`, and it contains an array of `PacketChunk` objects describing the individual elements of the UMP message.

After the app launches, it creates a `PacketSender` object that the app uses to construct and send MIDI packets. The first step of initialization is creating the Core MIDI client the app maintains as a strong reference throughout its lifetime. Next, the app creates an output port by calling `MIDIOutputPortCreate(_:_:_:)`.

```
private func setupMIDI() -> Bool {
    let status = MIDIClientCreateWithBlock("Packet Sender" as CFString, &client, { [
        self?.handleMIDI(notification)
    })
    guard status == noErr else {
        print("Failed to create the MIDI client.")
        return false
    }

    if midiAdapter.openMIDIPort(client, named: "MIDI Output Port" as CFString, port:
        print("Failed to create the MIDI port.")
        return false
    }
    return true
}
```

After the user presses the Send button, the sample gets the current values for each `PacketModel` and — depending on the MIDI protocol — creates a `MIDIMessage_32` or `MIDIMessage_64` by using one of the MIDI Messages.

The code below shows how the app sends a note on an event for the MIDI-1UP and MIDI 2 protocols. In the `noteOn(._1_0)` case, the app calls `MIDI1UPNoteOn(_:_:_:_:)` to create a 32-bit message, and passes in a MIDI group, channel, note number, and velocity. In the `noteOn(._2_0)` case, the app calls `MIDI2NoteOn(_:_:_:_:_:_:)` to create a 64-bit message, and passes in a MIDI group, channel, note number, attribute type, attribute data, and velocity.

```
case .noteOn(._1_0):
    guard let noteNumber = packetModel.getChunk(PacketChunkDescription.noteNumber)?.
          let velocity = packetModel.getChunk(PacketChunkDescription.velocity)?.uint
                return
            }

    result = midiAdapter.sendMIDI1UPMessage(MIDI1UPNoteOn(group,
                                                          channel,
                                                          noteNumber,
                                                          velocity),
                                            port: port,
                                            destination: destination)

case .noteOn(._2_0):
    guard let noteNumber = packetModel.getChunk(PacketChunkDescription.noteNumber)?.
          let attributeType = packetModel.getChunk(PacketChunkDescription.attributeT
          let attributeData = packetModel.getChunk(PacketChunkDescription.attribute)
          let velocity = packetModel.getChunk(PacketChunkDescription.velocity)?.uint
                return
            }

    result = midiAdapter.sendMIDI2Message(MIDI2NoteOn(group,
                                                      channel,
                                                      noteNumber,
                                                      attributeType,
                                                      attributeData,
                                                      velocity),
                                          port: port,
                                          destination: destination)
```

Send a MIDI event to a destination

The sample sends data by initializing a `MIDIEventList` and passing the event by reference, supplying the MIDI protocol as the second argument. Finally, the sample calls `MIDIEventList Add( : : : : : :)` to add the event to the list, and calls `MIDISendEventList( : : :)` with a port, destination, and the event list.

```
-(OSStatus)sendMIDI1UPMessage:(MIDIMessage_32)message port:(MIDIPortRef)port destina
    MIDIEventList eventList = {};
    MIDIEventPacket *packet = MIDIEventListInit(&eventList, kMIDIProtocol_1_0);
    packet = MIDIEventListAdd(&eventList, sizeof(MIDIEventList), packet, 0, 1, (UInt
    return MIDISendEventList(port, destination, &eventList);
}
```

Receive a MIDI event from a destination

The receiver target demonstrates how to create a Core MIDI destination with a protocol, and consumes incoming MIDI events, displaying them in a MIDI log.

The sample provides a text box to enter a custom destination name, a picker for protocol selection, and a log to view the incoming MIDI events.

The receiver target creates a Core MIDI client, and when the user presses the Create Destination button, the sample uses `MIDIDestinationCreateWithProtocol( : : : : :)` to create a destination with a name, a MIDI protocol, and a callback block. When the system receives events, the sample iterates the `MIDIEventList` and pushes them onto a queue to consume on the main thread.

```
-(OSStatus)createMIDIDestination:(MIDIClientRef)client named:(CFStringRef)name proto
    __block MIDIMessageFIFO *msgQueue = messageQueue.get();
    const auto status = MIDIDestinationCreateWithProtocol(client, name, protocol, ou

        if (evtlist->numPackets > 0 && msgQueue) {
            auto pkt = &evtlist->packet[0];

            for (int i = 0; i < evtlist->numPackets; ++i) {
                if (!msgQueue->push(evtlist->packet[i])) {
                    msgQueue->push(evtlist->packet[i]);
                }
                pkt = MIDIEventPacketNext(pkt);
            }
        }
    });
    return status;
```

```
}
```

> **Note**
>
> The system calls `MIDIReceiveBlock` on a high-priority thread, so perform real-time safe and nonblocking operations within the block.

See Also

Client management

`func MIDIClientCreate(CFString, MIDINotifyProc?, UnsafeMutableRawPointer?, UnsafeMutablePointer<MIDIClientRef>) -> OSStatus`

Creates a MIDI client.

`func MIDIClientCreateWithBlock(CFString, UnsafeMutablePointer<MIDIClientRef>, MIDINotifyBlock?) -> OSStatus`

Creates a MIDI client with a callback block.

`func MIDIClientDispose(MIDIClientRef) -> OSStatus`

Disposes of a MIDI client.

`typealias MIDIClientRef`

An object that maintains per-client state.