Musical Signal Processing with LabVIEW – MIDI for Synthesis and Algorithm Control

By:

Ed Doering

Musical Signal Processing with LabVIEW – MIDI for Synthesis and Algorithm Control

By:
Ed Doering

Online:
< http://cnx.org/content/col10487/1.2/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Chapter 1

MIDI Messages[1]

1.1 Introduction

A **MIDI message** conveys information between MIDI-capable equipment. For example, a message could indicate that a note should begin sounding, or that a specific type of sound be selected, or that the position of a pitch-bender control has just changed. MIDI messages are typically three bytes long: a **status byte** followed by two **data bytes**. Status and data bytes are distinguished by the value of the MSB (most-significant bit); the MSB is set to a "1" for status bytes, and is cleared to a "0" for data bytes.

When the MIDI standard was developed in 1983, MIDI messages were designed for compatibility with serial communications through **UART**s (universal asynchronous receiver-transmitters), the digital device "hiding" between the COM port on a desktop computer. In this way standard computer equipment could be interconnected to musical equipment including synthesizers, keyboards, sound modules, and drum machines.

The original MIDI electrical interconnection was designed for compatibility with standard audio cables terminated with **DIN-5** connectors. Look at the back of a typical synthesizer, and you will see three connectors that look like this:

[1]This content is available online at <http://cnx.org/content/m15049/1.2/>.

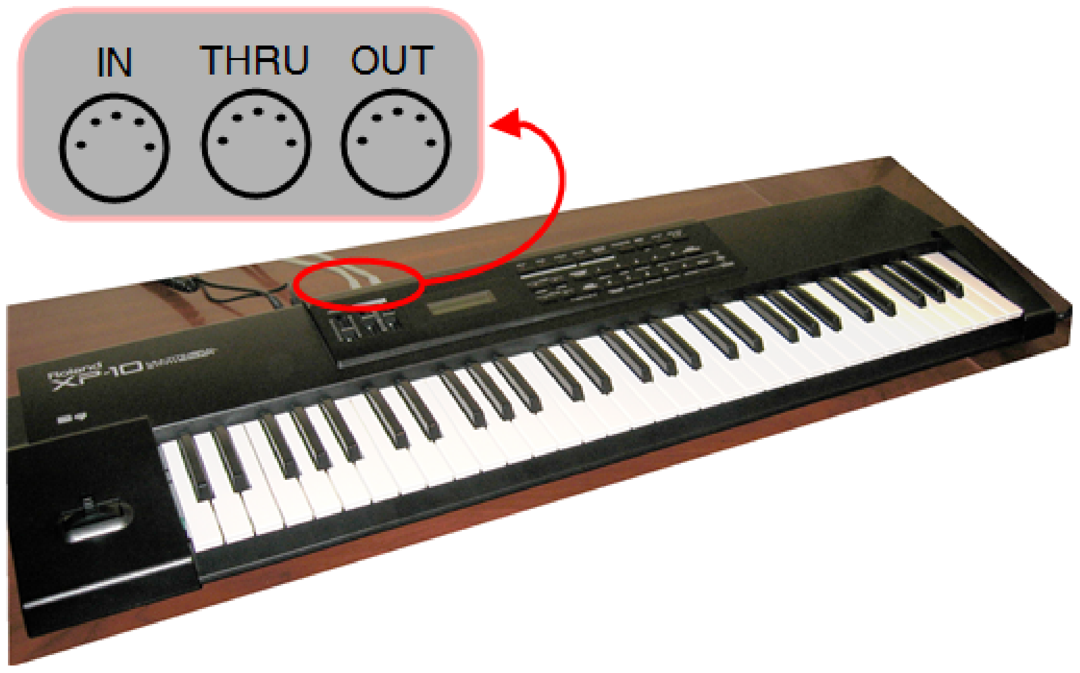


Figure 1.1: Rear-panel MIDI connectors for IN, OUT, and THRU

The electrical connection is unidirectional. The **MIDI IN** connector accepts a signal transmitted from the **MIDI OUT** connector of another device. Many different devices can be cabled together in a **daisy chain**, like this:

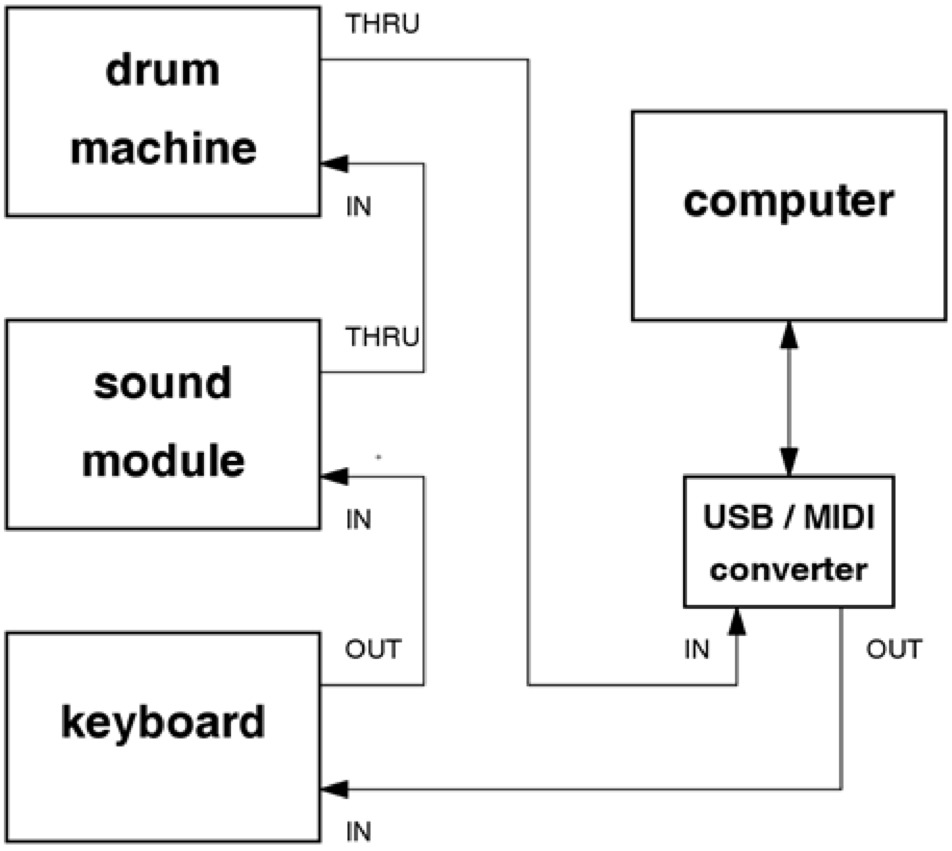


Figure 1.2: Typical daisy-chain connection scheme for several MIDI devices

Since it is not always desirable to have every single device process and re-transmit received MIDI messages, the **MIDI THRU** connector offers an electrically-identical version of the signal received on the **MIDI IN** connector. For example, the sound module and drum machine do not generate MIDI messages, so they can simply pass through the signals. However, the keyboard generates messages, so the **MIDI OUT** connector must be used.

MIDI messages associated with the actual musical performance – note on, note off, voice selection, and controller status – use the concept of a channel. For example, suppose that one synthesizer has been configured to associate "Channel 3" with the sound of a cello, and another synthesizer has been configured to associate "Channel 7" with the sound of a flute. When both of these interconnected synthesizers receive a "note on" message for Channel 3, only the first synthesizer will begin to sound a cello; the second synthesizer will ignore the message. In this way, many different devices can be interconnected, and configured to respond individually according to the channel number.

The MIDI standard for electrical interconnection specifies a fixed bit rate of 31.25 kbits/second. In the days of 8-MHz personal computers, 31.25 kbaud was considered quite fast. The rate is adequate to communicate performance information between several interconnected devices without noticeable delay, or **latency**. Today, however, MIDI messages are more often conveyed through **USB**. MIDI-to-USB converter boxes are available for older synthesizers that do not support USB directly.

1.2 Making a Sound: Note-On and Note-Off Events

As mentioned earlier, most MIDI messages are three bytes in length: a status byte followed by two data bytes. The status byte for the **Note-On event** is **1001nnnn** (0x**9n** in hexadecimal), where nnnn indicates the **channel number**. Since a four-bit value selects channel, there are a total of 16 channels available. The channels are called "Channel 1" (nnnn=0000) to "Channel 16" (nnnn=1111).

The first data byte indicates the **note number** to begin sounding, and the second data byte indicates the **velocity** at which the key was pressed. Since the MSB (most-significant bit) of a data byte is zero by definition, note number and velocity are each 7-bit values in the range 0 to 127.

Hexadecimal notation is commonly used to describe MIDI messages. For example, the three-byte message 0x93 0x5C 0x42 indicates a Note On event on Channel 4 for note number 92 with velocity 66.

The **Note-Off event** is similar; its status byte is **1000nnnn** (0x**8n**). For example, the three-byte message 0x83 0x5C 0x35 indicates that note number 92 on Channel 4 should cease sounding, and that the key release velocity was 53. Synthesizers and sound modules generally equate keypress velocity with amplitude; however, they are less likely to use the release velocity. A "Note On" event with zero velocity is equivalent to a "Note Off" event.

The following screencast video shows MIDI Note On and Note Off messages produced by the Roland XP-10 synthesizer and visualized using the MIDI OX software application[2] . MIDI OX is a free MIDI utility that serves as a MIDI protocol analyzer. The video also shows how MIDI OX can play standard MIDI files such as instruments.mid[3] , a short multi-instrument composition that illustrates the concept of MIDI channels:

Image not finished

Figure 1.3: [video] Visualize MIDI note-on and note-off messages generated by the Roland XP-10 synthesizer

If you plan to start using MIDI OX right away, take a look at the following screencast that explains how to set up MIDI OX to work with a MIDI device such as a synthesizer, and how to set up your soundcard so that you can visualize the MIDI messages produced by a MIDI player such as Winamp or Windows Media Player. You will also need to install the support application called MIDI Yoke[4] , which serves as a virtual MIDI patch bay to interconnect all of the MIDI-related devices on your computer.

Image not finished

Figure 1.4: [video] Setting up MIDI-OX to view MIDI messages

[2]http://www.midiox.com/
[3]http://instruments.mid/
[4]http://www.midiox.com/

Image not finished

Figure 1.5: [video] Setting up your soundcard to work properly with MIDI-OX and a media player

1.3 Selecting a Voice: Program Change

The **Program Change** message selects which of 128 possible voices (also called sounds, tones, or patches) to associate with a particular channel. The status byte for the Program Change message is **1100nnnn** (0x**Cn**), where nnnn indicates the channel number. Only one data byte called the **program number** follows the status byte for this message type. For example, the two-byte message 0xC7 0x5F directs the synthesizer to use program number 95 for all subsequent Note-On and Note-Off messages directed to Channel 8.

The original MIDI standard did not constrain equipment manufacturers in what sounds or voices to associate with program numbers. However, as it soon became apparent that defining standard voices for the 128 program numbers would make it easier for composers to distribute multi-voice (or **multitimbral**) compositions with the expectation of the listener hearing what the composer had intended. The **General MIDI (GM) standard** defines 128 basic voices to be associated with the 128 program numbers. Moreover, Channel 10 is always defined as a percussion instrument under the General MIDI standard. See the GM Level 1 Sound Set[5] for a table of voices and percussion key map. Note that the seven-bit program number ranges from 0 to 127; the associated GM voice will range from 1 to 128. The following screencast video gives you a quick tour of the 128 General MIDI voices as they sound on my Roland XP-10 synthesizer.

Image not finished

Figure 1.6: [video] Tour of the General MIDI sound set as played by the Roland XP-10 synthesizer

The seven-bit program number can select from among a palette of 128 voices, yet musicians and composers find this value too limiting. Equipment manufacturers can easily provide thousands of voices thanks to the falling cost of solid-state memory.

With these facts in mind, it will be easier to understand why selecting a new voice for a channel can produce up to three distinct MIDI messages! The first two messages are Control Change messages, and the third message is a Program Change message. The status byte of a **Control Change** message is **1011nnnn** (0x**Bn**), where nnnn indicates the channel number. The first data byte following the status byte indicates the **controller number**, and the second data byte indicates the **controller value**. The Control Change message will be described more fully in the following section. When the Control Change message is used select a voice, the controller number can be either 0x00 or 0x20 to indicate that the second data byte is either the most significant byte or the least significant byte, respectively, of a combined 14-bit **bank select value**. An example will clarify these concepts.

Suppose that the following three MIDI messages are received by a synthesizer (each message is destined for Channel 1):

```
0xB0 0x00 0x2F
0xB0 0x20 0x38
```

[5]http://www.midi.org/about-midi/gm/gm1sound.shtml

`0xC0 0x0E`

The first message is a Control Change with control number 0x00, so the upper seven bits of the bank select value are 010_1111. The second message is also a Control Change, but the 0x20 control number signifies that the lower seven bits of the bank select value are 011_1000. Thus the first two messages communicate a single 14-bit bank select value of 01_0111_1011_1000. Lastly, the Program Change message indicates that program number 16 is to be used. With all three messages, it is possible to select a unique voice from among 2 raised to the 21st power possible voices (2,097,152 possible voices).

This clever scheme permits newer equipment with more than 128 voices to properly respond to the rich palette of possible voices, while older equipment that only supports the basic 128 voices defined by the General MIDI standard will simply ignore the two bank select messages and respond to the Program Change message. Equipment manufacturers therefore will often define tone variations on a theme defined by the program number. For example, since program number 1 defines "Acoustic Grand Piano," the bank select technique can be used to define many different types of acoustic grand pianos.

Figure 1.7: [video] Visualize "Bank Select" and "Program Change" MIDI messages that select different voices

1.4 Modifying a Sound: Control Change and Pitch Wheel Change

Musicians need to do more than simply turn a note on and off. To be expressive, a musician needs to be able to modify a sound already in progress. Synthesizers offer various knobs and sliders that can introduce pitch bends and vibrato, for example.

The **Control Change** message indicates that a control has just been changed. The status byte of a Control Change message is **1011nnnn** (0x**Bn**), where nnnn indicates the channel number. The first data byte following the status byte indicates the **controller number**, and the second data byte indicates the **controller value**. Since the controller number is seven bits, 128 controller types are possible. Typical controllers include modulation wheel (0x01), main volume (0x07), sustain pedal (0x40), and general-purpose controllers (0x50 to 0x53).

Musicians use a synthesizer's **pitch wheel** introduces to bend (temporarily raise or lower) the pitch of the entire keyboard. In order to provide more frequency resolution, the pitch wheel has a dedicated MIDI message; the two data bytes afford fourteen bits of resolution rather than the seven bits of resolution that are possible with a control change message.

The **Pitch Wheel Change** message indicates that the pitch wheel has just moved. The status byte of a Pitch Wheel Change message is **1110nnnn** (0x**En**), where nnnn indicates the channel number. The first data byte following the status byte is the lower seven bits of the overall 14-bit position value, and the second data byte contains the upper seven bits of the position value. The nominal value is 0x2000 when the pitch wheel is centered (data byte #1 = 0x00, data byte #2 = 0x40). Moving the pitch wheel to its lower (leftmost limit) produces a value of 0x0000, while moving it to its rightmost limit produces a value of 0x3FFF (data byte #1 = 0x7F, data byte #2 = 0x7F). Note that the Pitch Wheel Change message simply indicates the **position** of the pitch wheel; the synthesizer may interpret this position in various ways depending on other settings. For example, my synthesizer defaults to a whole step down when the pitch wheel is moved to its left limit, however, it can be adjusted to shift by an entire octave for the same amount of pitch wheel movement.

The following screencast illustrates the MIDI messages generated by the slider controls and pitch wheel.

Image not finished

Figure 1.8: [video] Visualize "Control Change" MIDI messages produced by a general-purpose slider, and the "Pitch Wheel Change" messages produced by the pitch-bender

1.5 Transferring Data: System Exclusive Messages

The **System Exclusive** (**SysEx**) message provides a mechanism to transfer arbitrary blocks of information between devices. For example, the complete configuration of a synthesizer can be uploaded to a computer to be retrieved at a later time. The term "exclusive" indicates that that information pertains only to a particular vendor's piece of equipment, and that the organization of the information is vendor-specific.

The transfer process begins with the **System Exclusive Start** message with status byte **11110000** (0x**F0**). The following data byte is the manufacturer ID. All following bytes must be data bytes in the sense that their most-significant bit is always zero. An arbitrary number of bytes may follow the manufacturer ID. The transfer process ends with the **System Exclusive End** message with status byte **11110111** (0x**F7**); no data bytes follow this status byte.

1.6 Other Messages

The messages discussed in this module are representative of what you will encounter when working with synthesizers and standard MIDI files, but are by no means a complete listing of all available messages. Refer to the MIDI Manufacturers Association[6] for full details on the MIDI standard; see the specific tables noted in the last section of this module.

1.7 For Further Study

- MIDI Manufacturers Association (MMA)[7]
- MMA: Tutorial on MIDI and Music Synthesis[8]
- MMA: Table 1: Summary of MIDI messages[9]
- MMA: Table 2: Expanded Messages List (Status Bytes)[10]
- MMA: Table 3: Summary of Control Change Messages (Data Bytes)[11]
- MMA: About General MIDI[12]
- MMA: General MIDI Level 1 Sound Set[13]
- Textbook: Francis Rumsey, MIDI Systems and Control, 2nd ed., Focal Press, 1994. See Tables 3.1 (MIDI messages summarized), 2.2 (MIDI note numbers related to the musical scale), and 2.4 (MIDI controller functions).

[6]http://www.midi.org/
[7]http://www.midi.org/
[8]http://www.midi.org/about-midi/tutorial/tutor.shtml
[9]http://www.midi.org/about-midi/table1.shtml
[10]http://www.midi.org/about-midi/table2.shtml
[11]http://www.midi.org/about-midi/table3.shtml
[12]http://www.midi.org/about-midi/gm/gminfo.shtml
[13]http://www.midi.org/about-midi/gm/gm1sound.shtml

Chapter 2

Standard MIDI Files[1]

2.1 Introduction

In the 1970s, analog synthesizers were either played live by a keyboards musician or by a hardware device called a **sequencer**. The sequencer could be programmed with a pattern of notes and applied continuously as a loop to the keyboard. In 1983 the **MIDI** (**Musical Instrument Digital Interface**) standard was released, and MIDI-capable synthesizers could be connected to personal computers; the IBM PC and Apple Macintosh were among the first PCs to be connected to synthesizers. Software applications were developed to emulate the behavior of hardware sequencers, so naturally these programs were called "sequencers."

A **sequencer application** records MIDI messages and measures the time intervals between each received message. When the song is played back, the sequencer transmits the MIDI messages to the synthesizer, with suitable delays to match the original measurements. Sequencers store related MIDI messages in **tracks**. For example, the musician can record a drum track, and then record a bass track while listening to the drum track, and continue in this fashion until a complete song is recorded with full instrumentation. The finished result is stored in **a standard MIDI file** (**SMF**), normally named with the **.mid** suffix. In recent years sequencing software has merged with audio recording software to create a **digital audio workstation,** or **DAW**. Audio and MIDI tracks may be easily recorded, edited, and produced in one seamless environment.

Standard MIDI files can be played by a number of non-sequencer applications such as Winamp and Windows Media Player. In the absence of a synthesizer, these applications send the MIDI messages directly to the MIDI synthesizer included on the computer's soundcard. With a sufficiently high-quality soundcard, a software synthesizer application can be used instead of conventional synthesizer hardware to produce high-quality music; a USB piano-style keyboard is the only additional piece of gear needed.

After completing this module you will understand the structure of a standard MIDI file, including fundamental concepts such as **files chunks**, **delta-times** in **variable-length format**, **running status**, and **meta-events**. See MIDI Messages (Chapter 1) to learn more about the types of messages that can be contained in a standard MIDI file.

2.2 High-Level Structure

A **standard MIDI file** contains two high-level types of information called **chunks**: a single **header chunk**, and one or more **track chunks**. The header chunk defines which of three possible file types is in force, the number of track chunks in the file, and information related to timing. The track chunk contains pairs of **delta times** and **events**; the delta time indicates the elapsed time between two Note-On messages, for example.

[1]This content is available online at <http://cnx.org/content/m15051/1.3/>.

A chunk begins with the **chunk type**, a four-byte string whose value is either `MThd` for a header chunk or `MTrk` for a track chunk. The **chunk length** follows this string, and indicates the number of bytes remaining in the chunk. The chunk length is a 32-bit unsigned integer in **big-endian** format, i.e., the most-significant byte is first.

2.3 Header Chunk

The **header chunk** begins with the string `MThd`, and is followed by a fixed **chunk length** value of 6; as a 32-bit unsigned integer this is 0x00_00_00_06.

The **file type** follows, and is a 16-bit unsigned integer that takes on one of three possible values: 0 (0x00_00) indicates single track data, 1 (0x00_01) indicates multi-track data which is vertically synchronous (i.e., they are parts of the same song), and 2 (0x00_02) indicates multi-track data with no implied timing relationship between tracks. Type 1 is by far the most common, where each track contains MIDI messages on a single channel. In this way tracks can be associated with a single instrument and recorded individually

Next is the **number of tracks**, a 16-bit unsigned integer. The number of tracks can range from one (0x00_01) to 65,535 (0xFF_FF). In practice the number of tracks is typically about 20 or so.

The last value in the header chunk is the **division**, a 16-bit unsigned integer that indicates the number of **ticks per quarter note**. The **tick** is a dimensionless unit of time, and is the smallest grain of time used to indicate the interval between events. A typical value of division is 120 (0x00_78).

2.4 Track Chunk

The **track chunk** begins with the string `MTrk`, and is followed by the **track length**, a 32-bit unsigned integer that indicates the number of bytes remaining in the track. In theory a track could be as long as 4 Gbytes!

The remainder of the track is composed of pairs of delta-times and events. A **delta-time** is in units of tics, and indicates the time interval between events. An **event** is either a MIDI message or a **meta-event**. Meta-events are unique to the standard MIDI file, and indicate information such as track name, tempo, copyright notice, lyric text, and so on.

A delta time uses **variable-length format** (**VLF**), and can be anywhere from one to four bytes in length. Short delta times require only one byte, and long delta times can require up to four bytes. Since short delta times tend to dominate the standard MIDI file (think of a chord hit that generates a burst of Note-On messages with very little time between events), most of the delta times are only one byte long and the overall file size is thus minimized. However, since very long delta times must be accommodated as well, the variable-length format can use up to four bytes to represent very long time intervals.

2.5 Variable-Length Format

Variable-length format (**VLF**) is used to represent delta times and the length of meta-events (to be described below). A numerical value represented in VLF requires from one to four bytes, depending on the size of the numerical value. Since the standard MIDI file is parsed one byte at a time, some type of scheme is required to let the parser know the length of the VLF number. A naïve approach would be to include an additional byte at the beginning of the VLF to indicate the number of bytes remaining in the value. However, this approach would mean that a delta time always requires a minimum of two bytes. Instead, the VLF uses the most-significant bit (MSB) as a flag to indicate whether more bytes follow.

For example, consider the unsigned integer 0x42 = 0b0100_0010 (66 decimal), which can fit within seven bits. Since the MSB is clear (is zero), the parser will not seek any further bytes, and will conclude that the numerical value is 0x42.

Now consider the slightly larger value 0x80 = 0b1000_0000 (128 in decimal), which requires all eight bits of the byte. Since a VLF byte only has seven usable bits, this value must be encoded in two bytes, like this: 0x81 0x00. Why? If we group the original value into two 7-bit groups, we have the two binary values 0b000_0001 and 0b000_0000. The MSB of the first group must be set to 1 to indicate that more bytes follow, so it becomes 0b1000_0001 = 0x81. The MSB of the second group must be set to 0 to indicate that no bytes follow, so it becomes 0b0000_0000 = 0x00.

Exercise 2.1 ***(Solution on p. 13.)***

A delta time in VLF appears as two bytes, 0x820C. What is the delta time expressed as a decimal value?

Exercise 2.2 ***(Solution on p. 13.)***

A delta time in VLF appears as four bytes, 0xF3E8A975. What is the delta time expressed as a hexadecimal value?

2.6 Running Status

Running status is another technique developed to minimize the size of a standard MIDI file. Consider a long sequence of Note-On and Note-Off messages ("long" meaning perhaps thousands of events). Eventually you would notice that storing the status byte for each and every note message would seem redundant and wasteful. Instead, it would be more economical to store one complete Note-On message (a status byte followed by the note number byte and the velocity byte), and then from that point onwards store **only** the note number byte and velocity byte.

Clearly this is a simple solution for the application that creates the MIDI file, but what about the application that must read and parse the file? How can the parser know whether or not the status byte has been omitted from a message? Fortunately, the status byte can easily be distinguished from the data bytes by examination of the MSB (most-significant bit). When the parser expects a new message to start and finds that the byte has its MSB cleared (set to zero), the parser recovers the status information by using the information from the most-recent complete message. That is, the status continues to be the same as whatever was indicated by the most recent status byte received. In this way, a series of Note On messages can be conveyed by only two bytes per message instead of three.

But how do notes get turned off without storing a Note-Off message? Fortunately a Note-On message with zero velocity is equivalent to a Note-Off event! Thus, once the running status is established as "note on," it is possible to turn notes on and off for as long as you like, with each event requiring only two bytes.

Any time another message type needs to be inserted (such as a Program Change), the running status changes to match the new type of message. Alternatively, when a different channel is required for a note message, a fresh Note-On status byte must be sent. For these reasons, most standard MIDI files are organized as Type 1 (multi-track, vertically synchronous), where each track corresponds to a different voice on its own channel. The Program Change message occur at the beginning of the track to select the desired voice, a complete Note-On message is sent, and running status is used for the duration of the track to send Note-On and Note-Off messages as two-byte values.

2.7 Meta-Events

Meta-events provide a mechanism for file-related information to be represented, such as track name, tempo, copyright notice, lyric text, and so on. A meta-event begins with an 8-bit unsigned integer of value 0xFF. Note that the MSB (most-significant bit) is set, so a meta-event begins in the same way as a MIDI message status byte, whose MSB is also set. Next, the meta-event type is indicated by an 8-bit unsigned integer. After this, the number of bytes remaining in the meta-event is indicated by a numerical value in variable-length format (VLF); see an earlier section for full details). Lastly, the remainder of the meta-event information follows.

Some common meta-event types include 0x01 (**text event**), 0x02 (**copyright notice**), 0x03 (**track name**), 0x04 (**instrument name**), 0x05 (**lyric text**), and 0x7F (**sequencer-specific**). All of these meta-events can have an arbitrary length. Sequencer-specific is analogous to the **System-Exclusive** MIDI message, in that the data contained in the meta-event is arbitrary, and can normally be interpreted only by the sequencer application that created the standard MIDI file.

Tracks normally conclude with the **end-of-track** meta-event, whose type is 0x2F and whose length is zero. The end-of-track meta-event appears as the byte sequence 0xFF 0x2F 0x00 (meta-event, end-of-track type, zero length, with no data following).

The **set-tempo** meta-event (type 0x51) provides the value **microseconds per quarter note**, a 24-bit (3-byte) unsigned integer. This value in conjunction with the division value in the header chunk (division = ticks per quarter note) determines how to translate a delta time in ticks to a time in seconds. If the set-tempo meta-event is not included in the standard MIDI file, the value defaults to 500,000 microseconds per quarter note (or 0.5 seconds per quarter note).

Exercise: Given T (the value of a set-tempo meta-event) and D (the division value in the header chunk), determine an equation that can convert a delta time in ticks to a delta time in seconds.

2.8 For Further Study

- Description of MIDI Standard File Format[2]
- Standard MIDI Files 1.0[3]
- The USENET MIDI Primer[4] by Bob McQueer
- Outline of the Standard MIDI File Structure[5]
- Textbook: Francis Rumsey, MIDI Systems and Control, 2nd ed., Focal Press, 1994. See Table 5.2 (A selection of common meta-event type identifiers)

[2]http://www.4front-tech.com/pguide/midi/midi7.html
[3]http://jedi.ks.uiuc.edu/~johns/links/music/midifile.html
[4]http://www.sfu.ca/sca/Manuals/247/midi/UseNet-MidiPrimer.html
[5]http://www.ccarh.org/courses/253/handout/smf/

Solutions to Exercises in Chapter 2

Solution to Exercise 2.1 (p. 11)
268 [watch a video of the solution process[6]]
Solution to Exercise 2.2 (p. 11)
E7A14F5 [watch a video of the solution process[7]]

[6]http://cnx.org/content/m15051/latest/midi_VLF-ex1.html
[7]http://cnx.org/content/m15051/latest/midi_VLF-ex2.html

Chapter 3

Useful MIDI Software Utilities[1]

3.1 mf2t / t2mf

Standard MIDI files are binary files, and therefore cannot by read using a standard text editor. Piet van Oostrum has developed a companion pair of console applications called **mf2t** (MIDI File to Text) and **t2mf** (Text to MIDI File) that translate back and forth between the standard MIDI file and a human-readable version. You can more easily study the text-version MIDI file to see the messages, meta-events, and timing information. Also, you can edit the text-version, and then convert it back to the standard binary format.

Available at http://www.midiox.com[2] (scroll towards bottom of the page).

Image not finished

Figure 3.1: [video] Tour of the mf2t / t2mf MIDI-to-text conversion utilties

3.2 XVI32

Created by Christian Maas, **XVI32** is an excellent binary file editor (or "hex editor") when you need to view a standard MIDI file directly. XVI32 can also modify the file by tweaking individual byte values, or by inserting and deleting ranges of values. The editor also includes tools to interpret data values, i.e., select and range of 8 bytes and interpret as an IEEE double-precision floating point value.

Available at http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm[3]

Image not finished

Figure 3.2: [video] Tour of the XVI32 hex editor for viewing standard MIDI files

[1]This content is available online at <http://cnx.org/content/m14879/1.2/>.
[2]http://www.midiox.com/
[3]http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm

3.3 MIDI-OX

MIDI-OX is a wonderful utility developed Jamie O'Connell and Jerry Jorgenrud. MIDI-OX serves as a MIDI protocol analyzer by displaying MIDI data streams in real-time. MIDI-OX can also filter MIDI streams (remove selected message types) and map MIDI streams (change selected message types according to some rule). MIDI-OX includes other useful features: you can use your computer's keyboard to generate note events (you can even hold down multiple keys to play chords!), you can play standard MIDI files (.mid files), and you can capture a MIDI data stream and save it to a file.

In order to make full use of MIDI-OX, you will also want to install the **MIDI-Yoke** driver. MIDI-Yoke works like a virtual **MIDI patch bay**, a device that connects MIDI inputs and outputs together. For example, you can connect the output of a MIDI sequencer application to MIDI-OX to view the MIDI messages, and then send the message to the MIDI synthesizer on your soundcard. When you have many MIDI-capable devices connected to your computer, MIDI-OX and MIDI-Yoke make it easy to quickly re-configure the virtual connections without changing any external cables.

Available at http://www.midiox.com[4]

Image not finished

Figure 3.3: [video] Tour of the MIDI-OX utility

3.4 JAZZ++

JAZZ++ is a **MIDI sequencer** created by Andreas Voss and Per Sigmond. JAZZ++ serves as a multi-track recorder and editor for MIDI-capable instruments, and also supports audio tracks. A musician can create a composition with full instrumentation by recording tracks one at a time. Performances such as a piano solo recorded by the sequencer can easily be edited to correct any mistakes. JAZZ++ is also a great way to graphically visualize the contents of a standard MIDI file.

Available at http://www.jazzware.com/zope[5]

Image not finished

Figure 3.4: [video] Tour of the JAZZ++ MIDI sequencer application

[4]http://www.midiox.com/
[5]http://www.jazzware.com/zope

Chapter 4

P

arse and analyze a standard MIDI file][mini-project] Parse and analyze a standard MIDI file[1]

4.1 Objective

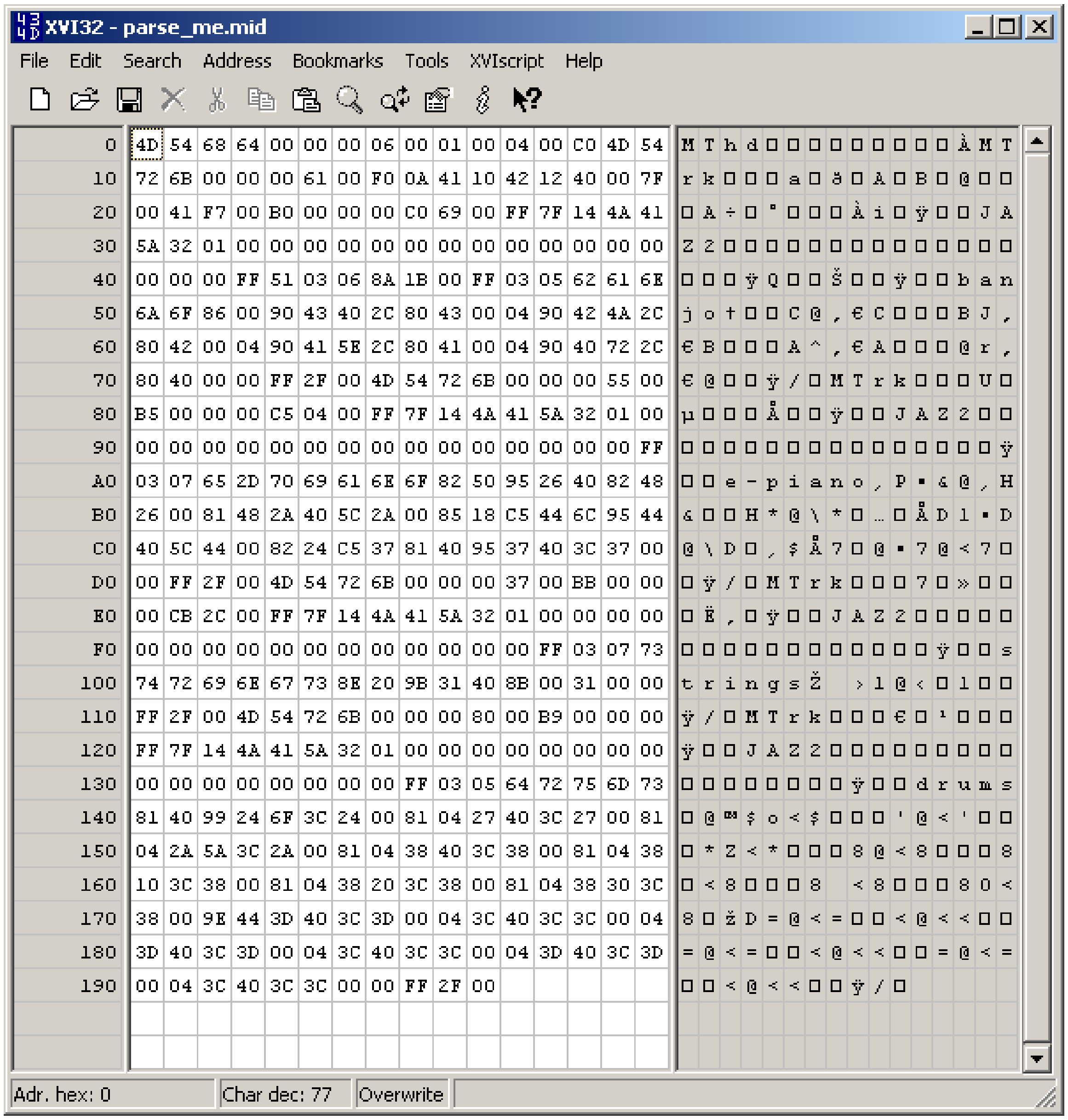
MIDI files consist of three types of data: (1) MIDI events, (2) timing information, and (3) file structure information (file header and track headers). Your task in this mini-project is to parse a MIDI file by hand by examining the individual bytes of the file. Once you can successfully parse a file, you will be well-positioned to write your own software applications that can create standard MIDI files directly, or even to read the file directly (a challenge!).

If you have not done so already, please study the pre-requisite modules, MIDI Messages (Chapter 1) and Standard MIDI Files (Chapter 2). You will need to refer to both of these modules in order to complete this activity.

4.2 Part 1: Parse the File

The file to be parsed is available here: parse_me.mid[2] (MIDI files are binary files, so right-click and choose "Save As" to download the file). If you wish, you can use a hexadecimal editor such as XVI32[3] to display the bytes, otherwise you can refer to the XVI32 screenshot below:

[1]This content is available online at <http://cnx.org/content/m15052/1.2/>.
[2]http://cnx.org/content/m15052/latest/parse_me.mid
[3]http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm



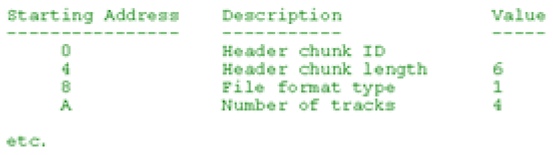
```
  0  4D 54 68 64 00 00 00 06 00 01 00 04 00 C0 4D 54
 10  72 6B 00 00 00 61 00 F0 0A 41 10 42 12 40 00 7F
 20  00 41 F7 00 B0 00 00 00 C0 69 00 FF 7F 14 4A 41
 30  5A 32 01 00 00 00 00 00 00 00 00 00 00 00 00 00
 40  00 00 00 FF 51 03 06 8A 1B 00 FF 03 05 62 61 6E
 50  6A 6F 86 00 90 43 40 2C 80 43 00 04 90 42 4A 2C
 60  80 42 00 04 90 41 5E 2C 80 41 00 04 90 40 72 2C
 70  80 40 00 00 FF 2F 00 4D 54 72 6B 00 00 00 55 00
 80  B5 00 00 00 C5 04 00 FF 7F 14 4A 41 5A 32 01 00
 90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF
 A0  03 07 65 2D 70 69 61 6E 6F 82 50 95 26 40 82 48
 B0  26 00 81 48 2A 40 5C 2A 00 85 18 C5 44 6C 95 44
 C0  40 5C 44 00 82 24 C5 37 81 40 95 37 40 3C 37 00
 D0  00 FF 2F 00 4D 54 72 6B 00 00 00 37 00 BB 00 00
 E0  00 CB 2C 00 FF 7F 14 4A 41 5A 32 01 00 00 00 00
 F0  00 00 00 00 00 00 00 00 00 00 00 00 FF 03 07 73
100  74 72 69 6E 67 73 8E 20 9B 31 40 8B 00 31 00 00
110  FF 2F 00 4D 54 72 6B 00 00 00 80 00 B9 00 00 00
120  FF 7F 14 4A 41 5A 32 01 00 00 00 00 00 00 00 00
130  00 00 00 00 00 00 00 00 FF 03 05 64 72 75 6D 73
140  81 40 99 24 6F 3C 24 00 81 04 27 40 3C 27 00 81
150  04 2A 5A 3C 2A 00 81 04 38 40 3C 38 00 81 04 38
160  10 3C 38 00 81 04 38 20 3C 38 00 81 04 38 30 3C
170  38 00 9E 44 3D 40 3C 3D 00 04 3C 40 3C 3C 00 04
180  3D 40 3C 3D 00 04 3C 40 3C 3C 00 04 3D 40 3C 3D
190  00 04 3C 40 3C 3C 00 00 FF 2F 00

Adr. hex: 0    Char dec: 77    Overwrite
```

Figure 4.1: Hexadecimal listing of the 'parse_me.mid' file

The byte position (address) is shown on the far left panel, the hexadecimal representation of each byte in the file is shown in the central panel, and the ASCII representation of each byte is shown in the right panel. Most of the ASCII characters look like gibberish, but important landmarks such as the track chunk ID (`MTrk`) are easily visible.

Interpret each part of the file, and list your results in tabular format, like this (use hexadecimal notation for the "Starting Address" column):



```
Starting Address    Description            Value
----------------    -----------            -----
     0              Header chunk ID
     4              Header chunk length      6
     8              File format type         1
     A              Number of tracks         4

etc.
```

Remember, a byte value of 0xFF signifies the beginning of a meta-event. Also, remember the concept of **running status**, so be on the lookout for MIDI messages whose first byte is NOT a status byte, i.e., whose MSB is clear. Take a look at the following video if you need some assistance:

Figure 4.2: [video] Walk through the MIDI file "parse_me.mid" at a high level, then show some parsing examples at the beginning of the file.

4.3 Part 2: Analyze Your Results

Determine the following information for the MIDI file:

- File format (Type 0, 1, or 2)
- Number of tracks
- Number of distinct channels used
- Number of distinct voices (instruments) used (look for Program Change messages)
- Number of ticks per quarter note
- Shortest two non-zero delta times, reported both in ticks and in microseconds (hint: the timing information you need here is derived from information in two distinct places)
- Top two longest delta times, both in ticks and in microseconds
- Minimum non-zero note-on velocity, and its channel and note number
- Maximum note-on velocity, and its channel and note number
- Total number of meta-events

If you have not done so already, listen to parse_me.mid[4] (click the link to launch your default music player application). How do your parsing and analysis results compare to what you hear?

[4]http://cnx.org/content/m15052/latest/parse_me.mid

Chapter 5

C

reate standard MIDI files with LabVIEW][mini-project] Create standard MIDI files with LabVIEW[1]

5.1 Required Background

If you have not done so already, please study the pre-requisite modules, MIDI Messages (Chapter 1) and Standard MIDI Files (Chapter 2). You will need to refer to both of these modules in order to complete this activity. Also, you will find it helpful to have already worked through the mini-project MIDI File Parsing (Chapter 4).

5.2 Introduction

In this project you will create your own LabVIEW application that can produce a standard MIDI file. You will first develop a library of six subVIs that can be combined into a top-level VI that operates just like **MIDI_UpDown.vi** below (click the "Run" button (right-pointing arrow) to create the MIDI file, then double-click on the MIDI file to hear it played by your soundcard):

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/MIDI_UpDown.llb

MIDI_UpDown.vi produces a two-track MIDI file, with one track an ascending chromatic scale and the other a descending chromatic scale. You can select the voice for each track by choosing a tone number in the range 1 to 128. You can also select the duration of each note ("on time") and space between the notes ("off time").

Remember, **MIDI_UpDown.vi** is simply a demonstration of a top-level VI constructed from the subVIs that you will make. Once you have constructed your library of subVIs, you will be able to use them in a wide array of projects; here are some ideas:

- simulated wind chime: with an appropriate voice selection (see the General MIDI Level 1 Sound Set[2] to choose a bell-like sound) and random number generator for delta times
- bouncing ping-pong ball: write a mathematical formula to model the time between bounces and the intensity of bounces
- custom ring-tone generator for a cell phone

[1]This content is available online at <http://cnx.org/content/m15054/1.2/>.
[2]http://www.midi.org/about-midi/gm/gm1sound.shtml

These are just a few ideas – be creative! Remember to take advantage of your ability to control the sound type, note-on velocity, pitch bend, etc.

5.3 Tour of the Top-Level Block Diagram

Click the image below to take a quick tour of the top-level block diagram of **MIDI_UpDown.vi**. The role of each subVI will be discussed in some detail, and you will have a better idea of the design requirements for each of the subVIs you will create.

Figure 5.1: [video] Tour of the top-level block diagram of MIDI_UpDown.vi

5.4 SubVI Library

You will create six subVIs in this part of the project. **Develop them in the exact order presented!** Also, make sure you **test and debug each subVI** before moving on to the next. Many of the concepts and techniques you learn at the beginning carry forward to the more sophisticated subVIs you develop toward the end.

The requirements for the subVIs are detailed in the following sections. **Input Requirements** specify the name of the front panel control, its data type, and default value, if needed). **Output Requirements** are similar, but refer to the front panel indicators. **Behavior Requirements** describe in broad terms the nature of the block diagram you need to design and build.

An interactive front panel is provided for each of the subVIs as an aid to your development and testing. By running the subVI with test values, you can better understand the behavioral requirements. Also, you can compare your finished result with the "gold standard," so to speak.

Screencast videos offer coding tips relevant to each subVI. The videos assume that you are developing the modules in the order presented.

All right, time to get to work!

NOTE: The file name convention adopted for this project will help you to better organize your work. Use the prefix "midi_" for the subVIs, and "MIDI_" for top-level applications that use the subVIs. This way all related subVIs will be grouped together when you display the files in the folder.

5.5 midi_PutBytes.vi

midi_PutBytes.vi accepts a string and writes it to a file. If the file already exists, the user should be prompted before overwriting the file.

5.5.1 Input Requirements

- file path (file path type)
- string (string type)

5.5.2 Output Requirements

- error out (error cluster)

5.5.3 Behavior Requirements

- Create a new file or replace an existing file
- If replacing a file, prompt the user beforehand to confirm
- Write the string to the file, then close the file
- Connect the file-related subVIs to `error out`

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see http://cnx.org/content/m15054/latest/midi_PutBytes.llb

5.5.4 Coding Tips

Watch the screencast video to learn how to use the built-in subVIs **Open/Create/Replace File**, **Write to Binary File**, and **Close File**. Refer to the module Creating a subVI in LabVIEW[3] to learn how to create a **subVI**.

Figure 5.2: [video] Learn how to write a string to a binary file

5.6 midi_AttachHeader.vi

Once all of the track strings have been created, **midi_AttachHeader.vi** will attach a header chunk to the beginning of the string to make a complete string prior to writing to a file. The header chunk requires the MIDI file type, number of tracks, and division (ticks per quarter note).

5.6.1 Input Requirements

- string in (string type)
- type (16-bit unsigned integer type; defaults to 1)
- number of tracks (16-bit unsigned integer type; defaults to 1)
- ticks per qnote (16-bit unsigned integer type; defaults to 120)

5.6.2 Output Requirements

- string out (string type)

[3]"Create a SubVI in LabVIEW" <http://cnx.org/content/m14767/latest/>

5.6.3 Behavior Requirements

- Create a header chunk ID sub-string (MThd)
- Create a sub-string for chunk length (always 0x00_00_00_06)
- Create sub-strings for the three unsigned integers applied as inputs
- Assemble the sub-strings into a string in order as chunk ID, chunk length, type, number of tracks, and division
- Append the inbound string to the header and output this result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see http://cnx.org/content/m15054/latest/midi_AttachHeader.llb

5.6.4 Coding Tips

Watch the screencast video to learn how to use the **Concatenate Strings** node to join substrings together into a single string. You will also learn how use the nodes **To Variant** and **Variant to Flattened String** to convert a numerical value into its representation as a sequence of bytes in a string.

Image not finished

Figure 5.3: [video] Learn how to concatenate strings and convert numerical values to a sequence of bytes

5.7 midi_FinishTrack.vi

Once all of the delta-time/event pairs have been assembled into a string, **midi_FinishTrack.vi** will attach a track chunk header to the beginning of the string and append an end-of-track meta-event at the end of the string. The resulting string will represent a complete track chunk.

5.7.1 Input Requirements

- string in (string type)
- delta-time / event pairs (string type)

5.7.2 Output Requirements

- string out (string type)

5.7.3 Behavior Requirements

- Create a track chunk ID sub-string (MTrk)
- Create a sub-string for a zero delta-time followed by an end-of-track meta-event
- Determine the total number of bytes in the track, and create a four-byte substring that represents this value
- Assemble the sub-strings into a string in order as chunk ID, chunk length, inbound string, and zero delta-time, and end-of-track meta-event, and output this result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/midi_FinishTrack.llb

5.7.4 Coding Tips

Watch the screencast video to learn how to use the nodes **String Length** and **To Unsigned Long Integer** to determine the number of bytes in the track.

Image not finished

Figure 5.4: [video] Learn how to determine the length of string

5.8 midi_ToVLF.vi

midi_ToVLF.vi accepts a 32-bit unsigned integer and produces an output string that is anywhere from one to four bytes in length (recall that VLF = variable length format). You may find this subVI to be one of the more challenging to implement! Review the module Standard MIDI Files (Chapter 2) to learn about variable-length format.

5.8.1 Input Requirements

- x (32-bit unsigned integer)
- string in (string type)

5.8.2 Output Requirements

- string out (string type)

5.8.3 Behavior Requirements

- Accept a numerical value to be converted into a sub-string one to four bytes in length in variable-length format
- Append the sub-string to the inbound string, and output the result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see http://cnx.org/content/m15054/latest/midi_ToVLF.llb

5.8.4 Coding Tips

Watch the screencast video to learn how to convert a numerical value to and from the **Boolean Array** data type, an easy way to work with values at the bit level.

Figure 5.5: [video] Learn how to convert a numerical value to a Boolean array in order to work at the individual bit level

5.9 midi_MakeDtEvent.vi

midi_MakeDtEvent.vi creates a delta-time / event pair. The subVI accepts a delta-time in ticks, a MIDI message selector, the channel number, and two data values for the MIDI message. The delta-time is converted into variable-length format, and the (typically) three-byte MIDI message is created. Both of these values are appended to the inbound string to produce the output string. Review the module MIDI Messages (Chapter 1) to learn more.

5.9.1 Input Requirements

- string in (string type)
- delta time (32-bit unsigned integer; defaults to 0)
- event (enumerated data type with values Note Off, Note On, Control Change, Program Change, Pitch Wheel; defaults to Note Off)
- channel (8-bit unsigned integer; defaults to 1)
- data 1 (8-bit unsigned integer; defaults to 0)
- data 2 (8-bit unsigned integer; defaults to 0)

5.9.2 Output Requirements

- string out (string type)

5.9.3 Behavior Requirements

- Convert the delta time value to VLF format using the **midi_ToVLF.vi** subVI you created in a previous step
- Subtract 1 from the inbound channel number (this way you can refer to channel numbers by their standard numbers (in the range 1 to 16) outside the subVI.
- Create a MIDI message status byte using the channel number and event selector
- Finish the MIDI message by appending the appropriate byte values; depending on the MIDI message you need to create, you may use both 'data 1' and 'data 2', or just 'data 1' (Program Change message), or you may need to modify the incoming data value slightly (for example, outside the subVI it is more convenient to refer to the tone number for a Program Change message as a value between 1 and 128)
- Append the delta-time sub-string and the MIDI message sub-string to the inbound string, and output the result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see http://cnx.org/content/m15054/latest/midi_MakeDtEvent.llb

5.9.4 Coding Tips

Watch the screencast video to learn how to assemble a byte at the bit level, and also how to set up the enumerated data type for a case structure.

Figure 5.6: [video] Learn how to assemble a byte at the bit level, and learn how to set up an enumerated data type for a case structure

5.10 midi_MakeDtMetaEvent.vi

midi_MakeDtMeta.vi creates a delta-time / meta-event pair. The subVI accepts a delta-time in ticks, a meta-event selector, text string, and tempo value (only certain meta-events require the last two inputs). The delta-time is converted into variable-length format, and the meta-event is created. Both of these values are appended to the inbound string to produce the output string. Review the module Standard MIDI Files (Chapter 2) to learn about meta-events.

5.10.1 Input Requirements

- string in (string type)
- delta time (32-bit unsigned integer; defaults to 0)
- event (enumerated data type with values Text, Copyright Notice Text, Track Name, Instrument Name, Lyric Text, Marker Text, Cue Point Text, Sequencer-Specific, End of Track, and Set Tempo; defaults to Track Name)
- text (string type)
- tempo (32-bit unsigned integer; defaults to 500,000)

5.10.2 Output Requirements

- string out (string type)

5.10.3 Behavior Requirements

- Convert the delta time value to VLF format using the **midi_ToVLF.vi** subVI you created in a previous step
- Create a meta-event sub-string using the sequence 0xFF (indicates meta-event), meta-event type (refer to a table of meta-event type numbers), meta-event length (use **midi_ToVLF.vi** for this purpose), and meta-event data.
- Append the delta-time sub-string and the MIDI message sub-string to the inbound string, and output the result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see http://cnx.org/content/m15054/latest/midi_MakeDtMetaEvent.llb

---

5.10.4 Coding Tips

At this point you should have enough experience to proceed without assistance!

5.11 Top-Level VI Application

Congratulations! You now have assembled and tested six subVIs that can form the basis of many other projects. Now use your subVIs to build an **application VI** (top-level VI) to demonstrate that your subVIs work properly together. Two applications that you can easily build are described next.

5.11.1 Sweep Through Notes and Tones

The application block diagram pictured below produces a single-track MIDI file containing an ascending chromatic sweep over the entire range of note numbers (0 to 127). Before sounding the note, the Program Number (tone or voice selection) is set to the same value as the note number. Thus, the voice changes for each note, adding additional interest to the sound. The note duration is specified by a control whose unit is milliseconds. As an exercise, you will need to complete the grayed-out area to convert the units of "duration" from milliseconds to ticks.

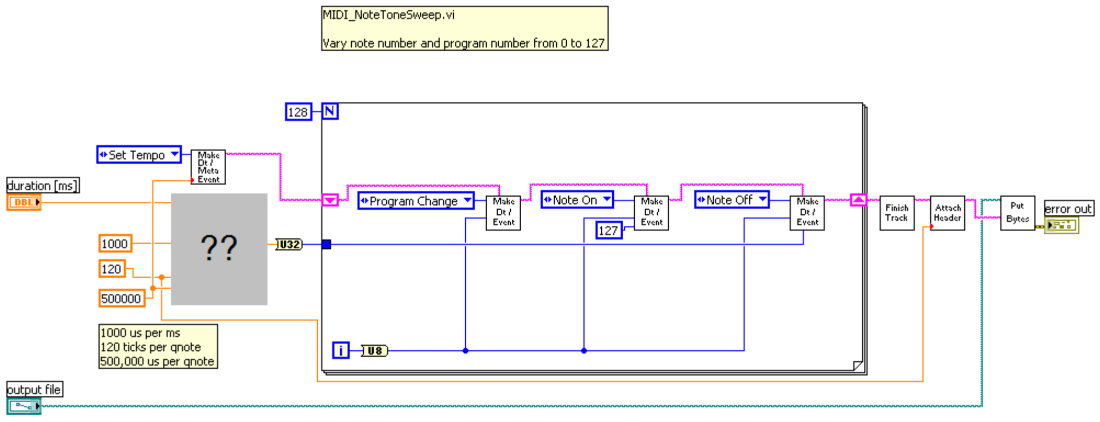


Figure 5.7: Block diagram to produce a single-track MIDI file containing an ascending chromatic sweep over the entire range of note numbers (0 to 127)

5.11.2 Measure the Velocity Profile of Your Soundcard

The application block diagram pictured below creates a MIDI file in which the same note is played repeatedly, but the velocity varies from the maximum to the minimum value in unit steps. When you play the MIDI file, you can record the soundcard's audio output and measure its **velocity profile**, i.e., the mapping between the note's velocity value and its waveform amplitude. The Audacity[4] sound editing application works well for this purpose; choose "Wave Out Mix" as the input device to record the soundcard's output.

[4]http://audacity.sourceforge.net/

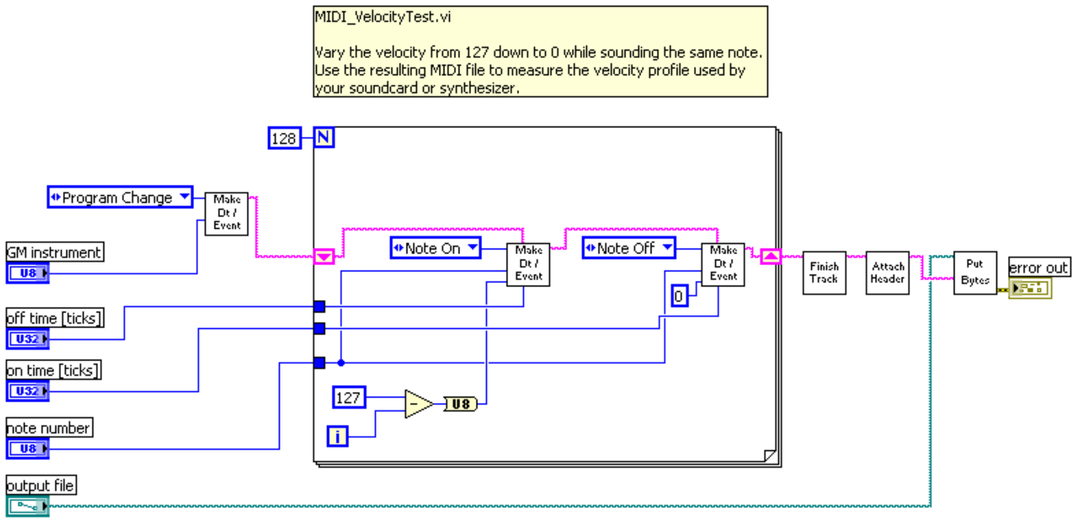


Figure 5.8: Block diagram to play a single note with velocity varied from 127 down to 0

Chapter 6

M

IDI_JamSession][LabVIEW application] MIDI_JamSession[1]

6.1 Introduction

MIDI_JamSession is a LabVIEW application VI that reads a standard MIDI file (.mid format) and renders it to a audio using "instrument" subVIs of your own design. Following are the key features of **MIDI_JamSession**:

- Reads standard MIDI files (.mid)
- Renders note events to stereo audio using user-defined subVIs called "virtual musical instruments" (VMIs) or built-in preview instruments
- Displays relevant MIDI file information to help determine how to assign instruments to MIDI channels
- Includes basic "mixing board" with controls for instrument type, mute, and stereo pan
- Creates files for rendered audio (.wav format) and note events (.csv spreadsheet format)

A MIDI file contains note and timing information (see MIDI Messages (Chapter 1) and Standard MIDI Files (Chapter 2) for full details). Notes are associated with **channels** (up to 16 channels possible). A single channel is almost always associated with a single instrument sound. **MIDI_JamSession.vi** uses all of this information to repeatedly call your **virtual musical instrument** (**VMI**) which creates a single note (an audio fragment) according to the requested duration, frequency, and amplitude; the audio fragment is then superimposed on the output audio stream at the correct time.

The following screencast video demonstrates how to use **MIDI_JamSession** to render MIDI files using the default preview instruments, and how to get started creating subVIs to render audio according to your own algorithms.

Figure 6.1: [video] Demonstration of the **MIDI_JamSession** application

[1]This content is available online at <http://cnx.org/content/m15053/1.2/>.

6.2 Source Distribution

MIDI_JamSession.vi is available in this .zip archive: MIDI_JamSession_v0.92.zip[2] . Right-click and choose "Save As" to download the .zip file, unpack the archive into its own folder, and double-click "MIDI_JamSession_run-me.vi" to start the application.

6.3 Instructions

- Start "MIDI_JamSession.vi" and choose a source MIDI file (.mid format); several MIDI files are included in the .zip distribution archive (see 'readme_midi-files.txt' for details). Click the folder icon to the right of the text entry browse to browse for a file. Once you select a file, "MIDI_JamSession" immediately reads the file and updates the MIDI information display panels. If you enter a filename in the "note events output file" field, a spreadsheet (in comma-separated values format) will be created that contains all of the note events extracted from the MIDI file. The columns are: channel number (1 to 16), start time (in seconds), duration (in seconds), note number (0 to 127), and velocity (0 to 127). The .csv file will be updated each time you load a new MIDI file.
- Leave all of the audio rendering controls at their default settings at first in order to use the built-in preview instruments, and to render only the first 10 percent of the song to audio. The relatively low sampling frequency and the simple algorithm for the preview instruments ensure quick rendering when you are exploring different MIDI files. Click "Render Audio" to listen to your MIDI file.
- If you have not done so already, double-click on your MIDI file to hear it played by your default media player using the built-in synthesizer on your computer's soundcard. "MIDI_JamSession" may not work properly for some types of MIDI files, so please compare the rendered audio to your media player's rendition before you continue.
- Look at the information text panels on the lower left, especially the track listing. Each channel number (inside square brackets) is typically associated with a unique instrument, and will often be labeled as such. The text entry boxes labeled "The Band" are where you assign your "virtual musical instrument" (VMI) to render notes for a given channel. Note that Channel 10 is reserved for percussion. The preview drum instrument renders all note events on Channel 10 the same way, regardless of note number or note velocity (it sounds a bit like a snare drum).
- Experiment with the pan controls to position each instrument in the stereo sound field; click "random pan" to make a random assignment for each instrument. You can also mute selected channels in order to isolate certain instruments, or to create a solo. Click the "Lock to 1" button to cause all controls to track those of Channel 1; this is an easy way to mute or unmute all channels, for example. Adjust the two sliders on the "time range to render" control to pick the start and stop times to render. You can quickly preview sections in the middle or end of the song this way. Set the controls to 0 and 100 percent to render the entire song.
- You will eventually find it more convenient to turn off the "Listen to audio" option and enter a filename in the "audio output file (.wav)" field. Each time you click "Render Audio" the .wav file will update, and you can use your own media player to listen to the .wav file. There is presently no way to interrupt the built-in audio player, and this can be a nuisance when you render long pieces. The yellow LED indicator at the upper right corner indicates when the built-in audio player is active.
- Once you are ready to create your own instrument sounds, open "vmi_Prototype.vi" and carefully follow the instructions inside. Eventually you will create a number of different VMIs, with each having the ability to generate an audio fragment that renders a single note.
- De-select the "Preview only" button, and select the VMI you wish to use for each channel in the vertical array of folders called "The Band." Blank entries will render as silence. Remember to adjust your sampling frequency as needed, bearing in mind that CD-quality (44.1 kHz) will increase the rendering time and increase the size of the .wav file.
- Render your new audio file.

[2]http://cnx.org/content/m15053/latest/MIDI_JamSession_v0.92.zip

- Enjoy listening!

IMPORTANT: Once you have invested a lot of effort to adjust the front panel settings, exit the application (click "Exit" just under the "MIDI Jam Session" logo), select "Edit | Make Current Values Default," and press Ctrl+S to save "MIDI_JamSession.vi" with your own settings.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Musical Signal Processing with LabVIEW – MIDI for Synthesis and Algorithm Control
The Musical Instrument Digital Interface (MIDI) standard specifies how to convey performance control information between synthesizer equipment and computers. Standard MIDI files (.mid extension) include timing information with MIDI messages to embody a complete musical performance. Learn about the MIDI standard, discover useful MIDI-related software utilities, and learn how to use LabVIEW to create a standard MIDI file according to your own design that can be played by any media appliance. Also learn about "MIDI JamSession," a LabVIEW application VI that renders standard MIDI files to audio using "virtual musical instruments" of your own design.