The MIDI Specification

MIDI consists of both a simple hardware interface, and a more elaborate transmission protocol.

Hardware

MIDI is an asynchronous serial interface. The baud rate is 31.25 Kbaud (+/- 1%). There is 1 start bit, 8 data bits, and 1 stop bit (ie, 10 bits total), for a period of 320 microseconds per serial byte.

The MIDI circuit is current loop, 5 mA. Logic 0 is current ON. One output drives one (and only one) input. To avoid grounding loops and subsequent data errors, the input is opto-isolated. It requires less than 5 mA to turn on. The Sharp PC-900 and HP 6N138 optoisolators are satisfactory devices. Rise and fall time for the optoisolator should be less than 2 microseconds.

The standard connector used for MIDI is a 5 pin DIN. Separate jacks (and cable runs) are used for input and output, clearly marked on a given device (ie, the MIDI IN and OUT are two separate DIN female panel mount jacks). 50 feet is the recommended maximum cable length. Cables are shielded twisted pair, with the shield connecting pin 2 at both ends. The pair is pins 4 and 5. Pins 1 and 3 are not used, and should be left unconnected.

A device may also be equipped with a *MIDI THRU* jack which is used to pass the MIDI IN signal to another device. The MIDI THRU transmission may not be performed correctly due to the delay time (caused by the response time of the opto-isolator) between the rising and falling edges of the square wave. These timing errors will tend to add in the "wrong direction" as more devices are daisy-chained to other device's MIDI THRU jacks. The result is that there is a limit to the number of devices that can be daisy-chained.

Messages

The MIDI protocol is made up of **messages**. A message consists of a string (ie, series) of 8-bit bytes. MIDI has many such defined messages. Some messages consist of only 1 byte.Other messages have 2 bytes. Still others have 3 bytes. One type of MIDI message can even have an unlimited number of bytes. The one thing that all messages have in common is that the first byte of the message is the **Status** byte. This is a special byte because it's the only byte that has bit #7 set. Any other following bytes in that message will not have bit #7 set. So, you can always detect the start a MIDI message because that's when you receive a byte with bit #7 set. This will be a Status byte in the range 0x80 to 0xFF. The remaining bytes of the message (ie, the data bytes, if any) will be in the range 0x00 to 0x7F.

The Status bytes of 0x80 to 0xEF are for messages that can be broadcast on any one of the 16 MIDI channels. Because of this, these are called **Voice** messages. (My own preference is to say that these messages belong in the **Voice Category**). For these Status bytes, you break up the 8-bit byte into 2 4-bit nibbles. For example, a Status byte of 0x92 can be broken up into 2 nibbles with values of 9 (high nibble) and 2 (low nibble). The high nibble tells you what *type* of MIDI message this is. Here are the possible values for the high nibble, and what type of Voice Category message each represents:

8 = Note Off
9 = Note On

A = AfterTouch (ie, key pressure)
B = Control Change
C = Program (patch) change
D = Channel Pressure
E = Pitch Wheel

So, for our example status of 0x92, we see that its message type is *Note On* (ie, the high nibble is 9). What's the low nibble of 2 mean? This means that the message is on MIDI channel 2. There are 16 possible (logical) MIDI channels, with 0 being the first. So, this message is a Note On on channel 2. What status byte would specify a *Program Change* on channel 0? The high nibble would need to be C for a Program Change type of message, and the low nibble would need to be 0 for channel 0. Thus, the status byte would be 0xC0. How about a Program Change on channel 15 (ie, the last MIDI channel). Again, the high nibble would be C, but the low nibble would be F (ie, the hexademical digit for 15).Thus, the status would be 0xCF.

NOTE: Although the MIDI Status byte counts the 16 MIDI channels as numbers 0 to F (ie, 15), all MIDI gear (including computer software) displays a channel number to the musician as 1 to 16. So, a Status byte sent on MIDI channel 0 is considered to be on "channel 1" as far as the musician is concerned. This discrepancy between the status byte's channel number, and what channel the musician "believes" that a MIDI message is on, is accepted because most humans start counting things from 1, rather than 0.

The Status bytes of 0xF0 to 0xFF are for messages that aren't on any particular channel (and therefore all daisy-chained MIDI devices always can "hear" and choose to act upon these messages. Contrast this with the Voice Category messages, where a MIDI device can be set to respond to those MIDI messages only on a specified channel). These status bytes are used for messages that carry information of interest to all MIDI devices, such as syncronizing all playback devices to a particular time. (By contrast, Voice Category messages deal with the individual musical parts that each instrument might play, so the channel nibble scheme allows a device to respond to its own MIDI channel while ignoring the Voice Category messages intended for another device on another channel).

These status bytes are further divided into two catagories. Status bytes of 0xF0 to 0xF7 are called System Common messages. Status bytes of 0xF8 to 0xFF are called System Realtime messages. The implications of such will be discussed later.

Actually, certain Status bytes within this range are not defined by the MIDI spec to date, and are reserved for future use. For example, Status bytes of 0xF4, 0xF5, 0xF9, and 0xFD are not used. If a MIDI device ever receives such a Status, it should ignore that message. See Ignoring MIDI Messages.

What follows is a description of each message type. The description tells what the message does, what its status byte is, and whether it has any subsequent data bytes and what information those carry. Generally, these descriptions take the view of a device receiving such messages (ie, what the device would typically be expected to do when receiving particular messages). When applicable, remarks about a device that transmits such messages may be made.

Note Off

Category: Voice

Purpose

Indicates that a particular note should be released. Essentially, this means that the note stops sounding, but some patches might have a long VCA release time that needs to slowly fade the sound out. Additionally, the device's Hold Pedal controller may be on, in which case the note's release is postponed until the Hold Pedal is released. In any event, this message either causes the VCA to move into the release stage, or if the Hold Pedal is on, indicates that the note should be released (by the device automatically) when the Hold Pedal is turned off. If the device is a MultiTimbral unit, then each one of its Parts may respond to Note Offs on its own channel. The Part that responds to a particular Note Off message is the one assigned to the message's MIDI channel.

Status

0x80 to 0x8F where the low nibble is the MIDI channel.

Data

Two data bytes follow the Status.

The first data is the note number. There are 128 possible notes on a MIDI device, numbered 0 to 127 (where Middle C is note number 60). This indicates which note should be released.

The second data byte is the velocity, a value from 0 to 127. This indicates how quickly the note should be released (where 127 is the fastest). It's up to a MIDI device how it uses velocity information. Often velocity will be used to tailor the VCA release time. MIDI devices that can generate Note Off messages, but don't implement velocity features, will transmit Note Off messages with a preset velocity of 64.

Errata

An All Notes Off controller message can be used to turn off all notes for which a device received *Note On* messages (without having received respective Note Off messages).

Note On

Category: Voice

Purpose

Indicates that a particular note should be played. Essentially, this means that the note starts sounding, but some patches might have a long VCA attack time that needs to slowly fade the sound in. In any case, this message indicates that a particular note should start playing (unless the velocity is 0, in which case, you really have a **Note Off**). If the device is a MultiTimbral unit, then each one of its Parts may sound Note Ons on its own channel. The Part that sounds a particular Note On message is the one assigned to the message's MIDI channel.

Status

0x90 to 0x9F where the low nibble is the MIDI channel.

Data

Two data bytes follow the Status.

The first data is the note number. There are 128 possible notes on a MIDI device, numbered 0 to 127 (where Middle C is note number 60). This indicates which note should be played.

The second data byte is the velocity, a value from 0 to 127. This indicates with how much force the note should be played (where 127 is the most force). It's up to a MIDI device how it uses velocity information. Often velocity is be used to tailor the VCA attack time and/or attack level (and therefore the overall volume of the note). MIDI devices that can generate Note On messages, but don't implement velocity features, will transmit Note On messages with a preset velocity of 64.

A Note On message that has a velocity of 0 is considered to actually be a Note Off message, and the respective note is therefore released. See the *Note Off* entry for a description of such. This "trick" was created in order to take advantage of *running status*.

A device that recognizes MIDI Note On messages **must** be able to recognize both a real Note Off as well as a Note On with 0 velocity (as a Note Off). There are many devices that generate real Note Offs, and many other devices that use Note On with 0 velocity as a substitute.

Errata

In theory, every Note On should eventually be followed by a respective Note Off message (ie, when it's time to stop the note from sounding). Even if the note's sound fades out (due to some VCA envelope decay) before a Note Off for this note is received, at some later point a Note Off should be received. For example, if a MIDI device receives the following Note On:

```
0x90 0x3C 0x40   Note On/chan 0, Middle C, velocity could be anything except 0
```

Then, a respective Note Off should subsequently be received at some time, as so:

```
0x80 0x3C 0x40   Note Off/chan 0, Middle C, velocity could be anything
```

Instead of the above Note Off, a Note On with 0 velocity could be substituted as so:

```
0x90 0x3C 0x00   Really a Note Off/chan 0, Middle C, velocity must be 0
```

If a device receives a Note On for a note (number) that is already playing (ie, hasn't been turned off yet), it is up the device whether to layer another "voice" playing the same pitch, or cut off the voice playing the preceding note of that same pitch in order to "retrigger" that note.

Aftertouch

Category: Voice

Purpose

While a particular note is playing, pressure can be applied to it. Many electronic keyboards have pressure sensing circuitry that can detect with how much force a musician is holding down a key. The musician can then vary this pressure, even while he continues to hold down the key (and the note continues sounding). The Aftertouch message conveys the amount of pressure on a key at a given point. Since the musician can be continually varying his pressure, devices that generate

Aftertouch typically send out many such messages while the musician is varying his pressure. Upon receiving Aftertouch, many devices typically use the message to vary a note's VCA and/or VCF envelope sustain level, or control LFO amount and/or rate being applied to the note's sound generation circuitry. But, it's up to the device how it chooses to respond to received Aftertouch (if at all). If the device is a MultiTimbral unit, then each one of its Parts may respond differently (or not at all) to Aftertouch. The Part affected by a particular Aftertouch message is the one assigned to the message's MIDI channel.

Status

0xA0 to 0xAF where the low nibble is the MIDI channel.

Data

Two data bytes follow the Status.

The first data is the note number. There are 128 possible notes on a MIDI device, numbered 0 to 127 (where Middle C is note number 60). This indicates to which note the pressure is being applied.

The second data byte is the pressure amount, a value from 0 to 127 (where 127 is the most pressure).

Errata

See the remarks under Channel Pressure.

Controller

Category: Voice

Purpose

Sets a particular controller's value. A controller is any switch, slider, knob, etc, that implements some function (usually) other than sounding or stopping notes (ie, which are the jobs of the Note On and Note Off messages respectively). There are 128 possible controllers on a MIDI device. These are numbered from 0 to 127. Some of these controller numbers as assigned to particular hardware controls on a MIDI device. For example, controller 1 is the *Modulation Wheel*. Other controller numbers are free to be arbitrarily interpreted by a MIDI device. For example, a drum box may have a slider controlling Tempo which it arbitrarily assigns to one of these free numbers. Then, when the drum box receives a Controller message for that controller number, it can adjust its tempo. A MIDI device need not have an actual physical control on it in order to respond to a particular controller.For example, even though a rack-mount sound module may not have a *Mod Wheel* on it, the module will likely still respond to and utilize *Modulation controller* messages to modify its sound. If the device is a MultiTimbral unit, then each one of its Parts may respond differently (or not at all) to various controller numbers. The Part affected by a particular controller message is the one assigned to the message's MIDI channel.

Status

0xB0 to 0xBF where the low nibble is the MIDI channel.

Data

Two data bytes follow the Status.

The first data is the controller number (0 to 127). This indicates which controller is affected by the received MIDI message.

The second data byte is the value to which the controller should be set, a value from 0 to 127.

Errata

An All Controllers Off controller message can be used to reset all controllers (that a MIDI device implements) to default values. For example, the *Mod Wheel* is reset to its "off" position upon receipt of this message.

See the list of Defined Controller Numbers for more information about particular controllers.

Program Change

Category: Voice

Purpose

To cause the MIDI device to change to a particular *Program* (which some devices refer to as Patch, or Instrument, or Preset, or whatever). Most sound modules have a variety of instrumental sounds, such as Piano, and Guitar, and Trumpet, and Flute, etc. Each one of these instruments is contained in a Program. So, changing the Program changes the instrumental sound that the MIDI device uses when it plays Note On messages. Of course, other MIDI messages also may modify the current Program's (ie, instrument's) sound. But, the Program Change message actually selects which instrument currently plays. There are 128 possible program numbers, from 0 to 127. If the device is a MultiTimbral unit, then it usually can play 16 "Parts" at once, each receiving data upon its own MIDI channel. This message will then change the instrument sound for only that Part which is set to the message's MIDI channel.

For MIDI devices that don't have instrument sounds, such as a Reverb unit which may have several Preset "room algorithms" stored, the Program Change message is often used to select which Preset to use. As another example, a drum box may use Program Change to select a particular rhythm pattern (ie, drum beat).

Status

0xC0 to 0xCF where the low nibble is the MIDI channel.

Data

One data byte follows the status. It is the program number to change to, a number from 0 to 127.

Errata

On MIDI sound modules (ie, whose Programs are instrumental sounds), it became desirable to define a standard set of Programs in order to make sound modules more compatible. This specification is called General MIDI Standard.

Just like with MIDI channels 0 to 15 being displayed to a musician as channels 1 to 16, many MIDI devices display their Program numbers starting from 1 (even though a Program number of 0 in a Program Change message selects the first program in the device). On the other hand, this approach was never standardized, and some devices use vastly different schemes for the musician to select a Program. For example, some devices require the musician to specify a bank of Programs, and then select one within the bank (with each bank typically containing 8 to 10 Programs). So, the musician might specify the first Program as being bank 1, number 1. Nevertheless, a Program Change of number 0 would select that first Program.

Channel Pressure

Category: Voice

Purpose

While notes are playing, pressure can be applied to all of them. Many electronic keyboards have pressure sensing circuitry that can detect with how much force a musician is holding down keys. The musician can then vary this pressure, even while he continues to hold down the keys (and the notes continue sounding). The Channel Pressure message conveys the amount of overall pressure on the keys at a given point. Since the musician can be continually varying his pressure, devices that generate Channel Pressure typically send out many such messages while the musician is varying his pressure. Upon receiving Channel Pressure, many devices typically use the message to vary all of the sounding notes' VCA and/or VCF envelope sustain levels, or control LFO amount and/or rate being applied to the notes' sound generation circuitry. But, it's up to the device how it chooses to respond to received Channel Pressure (if at all). If the device is a MultiTimbral unit, then each one of its Parts may respond differently (or not at all) to Channel Pressure. The Part affected by a particular Channel Pressure message is the one assigned to the message's MIDI channel.

Status

0xD0 to 0xDF where the low nibble is the MIDI channel.

Data

One data byte follows the Status. It is the pressure amount, a value from 0 to 127 (where 127 is the most pressure).

Errata

What's the difference between *AfterTouch* and *Channel Pressure*? Well, AfterTouch messages are for individual keys (ie, an Aftertouch message only affects that one note whose number is in the message). Every key that you press down generates its own AfterTouch messages. If you press on one key harder than another, then the one key will generate AfterTouch messages with higher values than the other key. The net result is that some effect will be applied to the one key more than the other key. You have individual control over each key that you play. With Channel Pressure, one message is sent out for the entire keyboard. So, if you press one key harder than another, the module will average out the difference, and then just pretend that you're pressing both keys with the exact same pressure. The net result is that some effect gets applied to all sounding keys evenly. You don't have individual control per each key. A controller normally uses either Channel Pressure or AfterTouch, but usually not both. Most MIDI controllers don't generate AfterTouch because that requires a pressure sensor for each individual key on a MIDI keyboard, and this is an expensive feature to implement. For this reason, many cheaper units implement Channel Pressure instead of

Aftertouch, as the former only requires one sensor for the entire keyboard's pressure. Of course, a device could implement both Aftertouch and Channel Pressure, in which case the Aftertouch messages for each individual key being held are generated, and then the average pressure is calculated and sent as Channel Pressure.

Pitch Wheel

Category: Voice

Purpose

To set the Pitch Wheel value. The pitch wheel is used to slide a note's pitch up or down in cents (ie, fractions of a half-step). If the device is a MultiTimbral unit, then each one of its Parts may respond differently (or not at all) to Pitch Wheel. The Part affected by a particular Pitch Wheel message is the one assigned to the message's MIDI channel.

Status

0xE0 to 0xEF where the low nibble is the MIDI channel.

Data

Two data bytes follow the status. The two bytes should be combined together to form a 14-bit value. The first data byte's bits 0 to 6 are bits 0 to 6 of the 14-bit value. The second data byte's bits 0 to 6 are really bits 7 to 13 of the 14-bit value. In other words, assuming that a C program has the first byte in the variable **First** and the second data byte in the variable **Second**, here's how to combine them into a 14-bit value (actually 16-bit since most computer CPUs deal with 16-bit, not 14-bit, integers):

```
unsigned short CombineBytes(unsigned char First, unsigned char Second)
{
unsigned short _14bit;

_14bit = (unsigned short)Second;
_14bit<<=7;
_14bit|=(unsigned short)First;
return(_14bit);
}
```

A combined value of 0x2000 is meant to indicate that the Pitch Wheel is centered (ie, the sounding notes aren't being transposed up or down). Higher values transpose pitch up, and lower values transpose pitch down.

Errata

The Pitch Wheel range is usually adjustable by the musician on each MIDI device. For example, although 0x2000 is always center position, on one MIDI device, a 0x3000 could transpose the pitch up a whole step, whereas on another device that may result in only a half step up. The GM spec recommends that MIDI devices default to using the entire range of possible Pitch Wheel message values (ie, 0x0000 to 0x3FFF) as +/- 2 half steps transposition (ie, 4 half-steps total range). The Pitch Wheel Range (or Sensitivity) is adjusted via an RPN controller message.

System Exclusive

Category: System Common

Purpose

Used to send some data that is specific to a MIDI device, such as a dump of its patch memory or sequencer data or waveform data. Also, SysEx may be used to transmit information that is particular to a device. For example, a SysEx message might be used to set the feedback level for an operator in an FM Synthesis device. This information would be useless to a sample playing device. On the other hand, virtually all devices respond to Modulation Wheel control, for example, so it makes sense to have a defined Modulation Controller message that all manufacturers can support for that purpose.

Status

Begins with 0xF0. Ends with a 0xF7 status (ie, after the data bytes).

Data

There can be any number of data bytes inbetween the initial 0xF0 and the final 0xF7. The most important is the first data byte (after the 0xF0), which should be a *Manufacturer's ID*.

Errata

Virtually every MIDI device defines the format of its own set of SysEx messages (ie, that only it understands). The only common ground between the SysEx messages of various models of MIDI devices is that all SysEx messages must begin with a 0xF0 status and end with a 0xF7 status.In other words, this is the only MIDI message that has 2 Status bytes, one at the start and the other at the end. Inbetween these two status bytes, any number of data bytes (all having bit #7 clear, ie, 0 to 127 value) may be sent. That's why SysEx needs a 0xF7 status byte at the end; so that a MIDI device will know when the end of the message occurs, even if the data within the message isn't understood by that device (ie, the device doesn't know exactly how many data bytes to expect before the 0xF7).

Usually, the first data byte (after the 0xF0) will be a defined *Manufacturer's ID*. The IMA has assigned particular values of the ID byte to various manufacturers, so that a device can determine whether a SysEx message is intended for it. For example, a Roland device expects an ID byte of 0x41. If a Roland device receives a SysEx message whose ID byte isn't 0x41, then the device ignores all of the rest of the bytes up to and including the final 0xF7 which indicates that the SysEx message is finished.

The purpose of the remaining data bytes, however many there may be, are determined by the manufacturer of a product. Typically, manufacturers follow the Manufacturer ID with a Model Number ID byte so that a device can not only determine that it's got a SysEx message for the correct manufacturer, but also has a SysEx message specifically for this model. Then, after the Model ID may be another byte that the device uses to determine what the SysEx message is supposed to be for, and therefore, how many more data bytes follow. Some manufacturers have a checksum byte, (usually right before the 0xF7) which is used to check the integrity of the message's transmission.

The 0xF7 Status byte is dedicated to marking the end of a SysEx message. It should never occur without a preceding 0xF0 Status. In the event that a device experiences such a condition (ie, maybe the MIDI cable was connected during the transmission of a SysEx message), the device should ignore the 0xF7.

Furthermore, although the 0xF7 is supposed to mark the end of a SysEx message, in fact, any status (except for Realtime Category messages) will cause a SysEx message to be considered "done". For example, if a 0x90 happened to be sent sometime after a 0xF0 (but before the 0xF7), then the SysEx message would be considered done at that point. It should be noted that, like all System Common messages, SysEx cancels any current running status. In other words, the next Voice Category message (after the SysEx message) must begin with a Status.

Here are the assigned Manufacturer ID numbers:

Sequential Circuits	1
Big Briar	2
Octave / Plateau	3
Moog	4
Passport Designs	5
Lexicon	6
Kurzweil	7
Fender	8
Gulbransen	9
Delta Labs	0x0A
Sound Comp.	0x0B
General Electro	0x0C
Techmar	0x0D
Matthews Research	0x0E
Oberheim	0x10
PAIA	0x11
Simmons	0x12
Gentle Electric	0x13
Fairlight	0x14
Peavey	0x1B
JL Cooper	0x15
Lowery	0x16
Lin	0x17
Emu	0x18
Bon Tempi	0x20
S.I.E.L.	0x21
SyntheAxe	0x23

Hohner	0x24
Crumar	0x25
Solton	0x26
Jellinghaus Ms	0x27
CTS	0x28
PPG	0x29
Elka	0x2F
Kawai	0x40
Roland	0x41
Korg	0x42
Yamaha	0x43
Casio	0x44
Akai	0x45

The following 2 IDs are dedicated to Universal SysEx messages (ie, SysEx messages that products from numerous manufacturers may want to utilize). Since SysEx is the only defined MIDI message that can have a length longer than 3 bytes, it became a candidate for using to transmit long strings of data. For example, many manufacturers make digital samplers. It became desirable for manufacturers to allow exchange of waveform data between each others' products. So, a standard protocol was developed called **MIDI Sample Dump Standard**. (SDS). Of course, since waveforms typically entail large amounts of data, SysEx messages (ie, containing over a hundred bytes each) seemed to be the most suitable vehicle to transmit the data over MIDI. But, it was decided not to use a particular manufacturer's ID. So, a universal ID was created. There's a universal ID meant for realtime messages (ie, ones that need to be responded to immediately), and one for non-realtime (ie, ones which can be processed when the device gets around to it).

RealTime ID	0x7F
Non-RealTime ID	0x7E

A general template for these two IDs was defined. After the ID byte is a :hp1.SysEx Channel:ehp1. byte. This could be from 0 to 127 for a total of 128 SysEx channels. So, although "normal" SysEx messages have no MIDI channel like Voice Category messages do, a Universal SysEx message can be sent on one of 128 SysEx channels. This allows the musician to set various devices to ignore certain Universal SysEx messages (ie, if the device allows the musician to set its Base SysEx Channel. Most devices just set their Base Sysex channel to the same number as the Base Channel for Voice Category messages). On the other hand, a SysEx channel of 127 is actually meant to tell the device to "disregard the channel and pay attention to this message regardless". After the SysEx channel, the next two bytes are *Sub IDs* which tell what the SysEx is for.

Besides the SDS messages (covered later in the SDS section), there are two other defined Universal Messages:

GM System Enable/Disable

This enables or disables the GM Sound module or GM Patch Set in a device. Some devices have built-in GM modules or GM Patch Sets in addition to non-GM Patch Sets or non-GM modes of operation. When GM is enabled, it replaces any non-GM Patch Set or non-GM mode. This allows a device to have modes or Patch Sets that go beyond the limits of GM, and yet, still have the capability to be switched into a GM-compliant mode when desirable.

```
    0xF0  SysEx

    0x7E  Non-Realtime

    0x7F  The SysEx channel. Could be from 0x00 to 0x7F.

          Here we set it to "disregard channel".

    0x09  Sub-ID -- GM System Enable/Disable

    0xNN  Sub-ID2 -- NN=00 for disable, NN=01 for enable

    0xF7  End of SysEx
```

Master Volume

This adjusts a device's master volume. Remember that in a multitimbral device, the *Volume controller* messages are used to control the volumes of the individual Parts. So, we need some message to control Master Volume. Here it is.

```
    0xF0  SysEx

    0x7F  Realtime

    0x7F  The SysEx channel. Could be from 0x00 to 0x7F.

          Here we set it to "disregard channel".

    0x04  Sub-ID -- Device Control

    0x01  Sub-ID2 -- Master Volume

    0xLL  Bits 0 to 6 of a 14-bit volume

    0xMM  Bits 7 to 13 of a 14-bit volume

    0xF7  End of SysEx
```

A manufacturer must get a registered ID from the IMA if he wants to define his own SysEx messages, or use the following:

```
Educational Use  0x7D
```

This ID is for educational or development use only, and should never appear in a commercial design.

On the other hand, it is permissible to use another manufacturer's defined SysEx message(s) in your own products. For example, if the Roland S-770 has a particular SysEx message that you could use verbatim in your own design, you're free to use that message (and therefore the Roland ID in it). But, you're not allowed to transmit a mutated version of any Roland message with a Roland ID. Only

Roland can develop new messages that contain a Roland ID.

MTC Quarter Frame Message

Category: System Common

Purpose

Some master device that controls sequence playback sends this timing message to keep a slave device in sync with the master.

Status

0xF1

Data

One data byte follows the Status. It's the time code value, a number from 0 to 127.

Errata

This is one of the MIDI Time Code (MTC) series of messages. See MIDI Time Code.

Song Position Pointer

Category: System Common

Purpose

Some master device that controls sequence playback sends this message to force a slave device to cue the playback to a certain point in the song/sequence. In other words, this message sets the device's "Song Position". This message doesn't actually start the playback. It just sets up the device to be "ready to play" at a particular point in the song.

Status

0xF2

Data

Two data bytes follow the status. Just like with the Pitch Wheel, these two bytes are combined into a 14-bit value. (See Pitch Wheel remarks). This 14-bit value is the *MIDI Beat* upon which to start the song. Songs are always assumed to start on a MIDI Beat of 0. Each MIDI Beat spans 6 *MIDI Clocks*. In other words, each MIDI Beat is a 16th note (since there are 24 MIDI Clocks in a quarter note).

Errata

Example: If a Song Position value of 8 is received, then a sequencer (or drum box) should cue playback to the third quarter note of the song. (8 MIDI beats * 6 MIDI clocks per MIDI beat = 48 MIDI Clocks. Since there are 24 MIDI Clocks in a quarter note, the first quarter occurs on a time of

0 MIDI Clocks, the second quarter note occurs upon the 24th MIDI Clock, and the third quarter note occurs on the 48th MIDI Clock).

Often, the slave device has its playback tempo synced to the master via MIDI Clock. See Syncing Sequence Playback.

Song Select

Category: System Common

Purpose

Some master device that controls sequence playback sends this message to force a slave device to set a certain song for playback (ie, sequencing).

Status

0xF3

Data

One data byte follows the status. It's the song number, a value from 0 to 127.

Errata

Most devices display "song numbers" starting from 1 instead of 0. Some devices even use different labeling systems for songs, ie, bank 1, number 1 song. But, a Song Select message with song number 0 should always select the first song.

When a device receives a Song Select message, it should cue the new song at MIDI Beat 0 (ie, the very beginning of the song), unless a subsequent Song Position Pointer message is received for a different MIDI Beat. In other words, the device resets its "Song Position" to 0.

Often, the slave device has its playback tempo synced to the master via MIDI Clock. See Syncing Sequence Playback.

Tune Request

Category: System Common

Purpose

The device receiving this should perform a tuning calibration.

Status

0xF6

Data

None

Errata

Mostly used for sound modules with analog oscillator circuits.

MIDI Clock

Category: System Realtime

Purpose

Some master device that controls sequence playback sends this timing message to keep a slave device in sync with the master. A MIDI Clock message is sent at regular intervals (based upon the master's Tempo) in order to accomplish this.

Status

0xF8

Data

None

Errata

There are 24 MIDI Clocks in every quarter note. (12 MIDI Clocks in an eighth note, 6 MIDI Clocks in a 16th, etc). Therefore, when a slave device counts down the receipt of 24 MIDI Clock messages, it knows that one quarter note has passed. When the slave counts off another 24 MIDI Clock messages, it knows that another quarter note has passed. Etc. Of course, the rate that the master sends these messages is based upon the master's tempo. For example, for a tempo of 120 BPM (ie, there are 120 quarter notes in every minute), the master sends a MIDI clock every 20833 microseconds. (ie, There are 1,000,000 microseconds in a second. Therefore, there are 60,000,000 microseconds in a minute. At a tempo of 120 BPM, there are 120 quarter notes per minute. There are 24 MIDI clocks in each quarter note. Therefore, there should be 24 * 120 MIDI Clocks per minute. So, each MIDI Clock is sent at a rate of 60,000,000/(24 * 120) microseconds).

A device might receive a Song Select message to cue a specific song to play (out of several songs), a Song Position Pointer message to cue that song to start on a particular beat, a MIDI Continue in order to start playback from that beat, periodic MIDI Clocks in order to keep the playback in sync with another sequencer, and finally a MIDI Stop to halt playback. See Syncing Sequence Playback.

MIDI Start

Category: System Realtime

Purpose

Some master device that controls sequence playback sends this message to make a slave device start playback of some song/sequence from the beginning (ie, MIDI Beat 0).

Status

0xFA

Data

None

Errata

A MIDI Start always begins playback at MIDI Beat 0 (ie, the very beginning of the song). So, when a slave device receives a MIDI Start, it automatically resets its "Song Position" to 0. If the device needs to start playback at some other point (either set by a previous Song Position Pointer message, or manually by the musician), then MIDI Continue is used instead of MIDI Start.

Often, the slave device has its playback tempo synced to the master via MIDI Clock. See Syncing Sequence Playback.

MIDI Continue

Category: System Realtime

Purpose

Some master device that controls sequence playback sends this message to make a slave device resume playback from its current "Song Position". The current Song Position is the point when the song/sequence was previously stopped, or previously cued with a Song Position Pointer message.

Status

0xFB

Data

None

Errata

Often, the slave device has its playback tempo synced to the master via MIDI Clock. See Syncing Sequence Playback.

MIDI Stop

Category: System Realtime

Purpose

Some master device that controls sequence playback sends this message to make a slave device stop playback of a song/sequence.

Status

0xFC

Data

None

Errata

When a device receives a MIDI Stop, it should keep track of the point at which it stopped playback (ie, its stopped "Song Position"), in the anticipation that a MIDI Continue might be received next.

Often, the slave device has its playback tempo synced to the master via MIDI Clock. See Syncing Sequence Playback.

Active Sense

Category: System Realtime

Purpose

A device sends out an Active Sense message (at least once) every 300 milliseconds if there has been no other activity on the MIDI buss, to let other devices know that there is still a good MIDI connection between the devices.

Status

0xFE

Data

None

Errata

When a device receives an Active Sense message (from some other device), it should expect to receive additional Active Sense messages at a rate of one approximately every 300 milliseconds, whenever there is no activity on the MIDI buss during that time. (Of course, if there are other MIDI messages happening at least once every 300 mSec, then Active Sense won't ever be sent. An Active Sense only gets sent if there is a 300 mSec "moment of silence" on the MIDI buss. You could say that a device that sends out Active Sense "gets nervous" if it has nothing to do for over 300 mSec, and so sends an Active Sense just for the sake of reassuring other devices that this device still exists). If a message is missed (ie, 0xFE nor any other MIDI message is received for over 300 mSec), then a device assumes that the MIDI connection is broken, and turns off all of its playing notes (which were turned on by incoming Note On messages, versus ones played on the local keyboard by a musician). Of course, if a device never receives an Active Sense message to begin with, it should not expect them at all. So, it takes one "nervous" device to start the process by initially sending out an Active Sense message to the other connected devices during a 300 mSec moment of silence on the MIDI bus.

This is an optional feature that only a few devices implement (ie, notably Roland gear). Many devices don't ever initiate this minimal "safety" feature.

Here's a flowchart for implementing Active Sense. It assumes that the device has a hardware timer that ticks once every millisecond. A variable named **Timeout** is used to count the passing milliseconds. Another variable named **Flag** is set when the device receives an Active Sense message from another device, and therefore expects to receive further Active Sense messages.

The logic for active sense detection

Reset

Category: System Realtime

Purpose

The device receiving this should reset itself to a default state, usually the same state as when the device was turned on. Often, this means to turn off all playing notes, turn the local keyboard on, clear running status, set Song Position to 0, stop any timed playback (of a sequence), and perform any other standard setup unique to the device. Also, a device may choose to kick itself into Omni On, Poly mode if it has no facilities for allowing the musician to store a default mode.

Status

0xFF

Data

None

Errata

A Reset message should never be sent automatically by any MIDI device. Rather, this should only be sent when a musician specifically tells a device to do so.

Controller Numbers

A *Controller* message has a Status byte of 0xB0 to 0xBF depending upon the MIDI channel. There are two more data bytes.

The first data byte is the *Controller Number*. There are 128 possible controller numbers (ie, 0 to 127). Some numbers are defined for specific purposes. Others are undefined, and reserved for future use.

The second byte is the "value" that the controller is to be set to.

Most controllers implement an effect even while the MIDI device is generating sound, and the effect will be immediately noticeable. In other words, MIDI controller messages are meant to implement various effects by a musician *while he's operating the device*.

If the device is a MultiTimbral module, then each one of its Parts may respond differently (or not at all) to a particular controller number. **Each Part usually has its own setting for every controller number**, and the Part responds only to controller messages on the same channel as that to which the Part is assigned. So, controller messages for one Part do not affect the sound of another Part even while that other Part is playing.

Some controllers are *continuous* controllers, which simply means that their value can be set to any value within the range from 0 to 16,384 (for 14-bit coarse/fine resolution) or 0 to 127 (for 7-bit, coarse resolution). Other controllers are *switches* whose state may be either **on** or **off**. Such controllers will usually generate only one of two values; 0 for off, and 127 for on. But, a device should be able to respond to any received switch value from 0 to 127. If the device implements only an "on" and "off" state, then it should regard values of 0 to 63 as off, and any value of 64 to 127 as on.

Many (continuous) controller numbers are *coarse* adjustments, and have a respective *fine* adjustment controller number. For example, controller #1 is the coarse adjustment for Modulation Wheel. Using this controller number in a message, a device's Modulation Wheel can be adjusted in large (coarse) increments (ie, 128 steps). If finer adjustment (from a coarse setting) needs to be made, then controller #33 is the fine adjust for Modulation Wheel. For controllers that have coarse/fine pairs of numbers, there is thus a 14-bit resolution to the range. In other words, the Modulation Wheel can be set from 0x0000 to 0x3FFF (ie, one of 16,384 values). For this 14-bit value, bits 7 to 13 are the coarse adjust, and bits 0 to 6 are the fine adjust. For example, to set the Modulation Wheel to 0x2005, first you have to break it up into 2 bytes (as is done with Pitch Wheel messages). Take bits 0 to 6 and put them in a byte that is the fine adjust. Take bits 7 to 13 and put them right-justified in a byte that is the coarse adjust. Assuming a MIDI channel of 0, here's the coarse and fine Mod Wheel controller messages that a device would receive (coarse adjust first):

```
0xB0 0x01 0x40

Controller on chan 0, Mod Wheel coarse, bits 7 to 13 of 14-bit

value right-justified (with high bit clear).



0xB0 0x33 0x05

Controller on chan 0, Mod Wheel fine, bits 0 to 6 of 14-bit

value (with high bit clear).
```

Some devices do not implement fine adjust counterparts to coarse controllers. For example, some devices do not implement controller #33 for Mod Wheel fine adjust. Instead the device only recognizes and responds to the Mod Wheel coarse controller number (#1). It is perfectly acceptable for devices to **only** respond to the coarse adjustment for a controller if the device desires 7-bit (rather than 14-bit) resolution. The device should ignore that controller's respective fine adjust message. By the same token, if it's only desirable to make fine adjustments to the Mod Wheel without changing its current coarse setting (or vice versa), a device can be sent only a controller #33 message without a preceding controller #1 message (or vice versa). Thus, if a device can respond to both coarse and fine adjustments for a particular controller (ie, implements the full 14-bit resolution), it should be able to deal with either the coarse or fine controller message being sent without its counterpart following. The same holds true for other continuous (ie, coarse/fine pairs of) controllers.

Here's a list of the defined controllers. To the left is the controller number (ie, how the MIDI

Controller message refers to a particular controller), and on the right is its name (ie, how a human might refer to the controller). To get more information about what a particular controller does, click on its controller name to bring up a description. Each description shows the controller name and number, what the range is for the third byte of the message (ie, the "value" data byte), and what the controller does. For controllers that have separate coarse and fine settings, both controller numbers are shown.

MIDI devices should use these controller numbers for their defined purposes, as much as possible. For example, if the device is able to respond to *Volume controller* (coarse adjustment), then it should expect that to be controller number 7. It should not use *Portamento Time controller* messages to adjust volume. That wouldn't make any sense. Other controllers, such as *Foot Pedal*, are more general purpose. That pedal could be controlling the tempo on a drum box, for example. But generally, the Foot Pedal shouldn't be used for purposes that other controllers already are dedicated to, such as adjusting *Pan position*. If there is not a defined controller number for a particular, needed purpose, a device can use the General Purpose Sliders and Buttons, or NRPN for device specific purposes. The device should use controller numbers 0 to 31 for coarse adjustments, and controller numbers 32 to 63 for the respective fine adjustments.

Defined Controllers

0	Bank Select
1	Modulation Wheel (coarse)
2	Breath controller (coarse)
4	Foot Pedal (coarse)
5	Portamento Time (coarse)
6	Data Entry (coarse)
7	Volume (coarse)
8	Balance (coarse)
10	Pan position (coarse)
11	Expression (coarse)
12	Effect Control 1 (coarse)
13	Effect Control 2 (coarse)
16	General Purpose Slider 1
17	General Purpose Slider 2
18	General Purpose Slider 3
19	General Purpose Slider 4
32	Bank Select (fine)
33	Modulation Wheel (fine)
34	Breath controller (fine)
36	Foot Pedal (fine)

37	Portamento Time (fine)
38	Data Entry (fine)
39	Volume (fine)
40	Balance (fine)
42	Pan position (fine)
43	Expression (fine)
44	Effect Control 1 (fine)
45	Effect Control 2 (fine)
64	Hold Pedal (on/off)
65	Portamento (on/off)
66	Sustenuto Pedal (on/off)
67	Soft Pedal (on/off)
68	Legato Pedal (on/off)
69	Hold 2 Pedal (on/off)
70	Sound Variation
71	Sound Timbre
72	Sound Release Time
73	Sound Attack Time
74	Sound Brightness
75	Sound Control 6
76	Sound Control 7
77	Sound Control 8
78	Sound Control 9
79	Sound Control 10
80	General Purpose Button 1 (on/off)
81	General Purpose Button 2 (on/off)
82	General Purpose Button 3 (on/off)
83	General Purpose Button 4 (on/off)
91	Effects Level
92	Tremulo Level
93	Chorus Level
94	Celeste Level

Bank Select

Number: 0 (coarse) 32 (fine)

Affects:

Some MIDI devices have more than 128 Programs (ie, Patches, Instruments, Preset, etc). MIDI Program Change messages only support switching between 128 programs. So, Bank Select Controller (sometimes called Bank Switch) is sometimes used to allow switching between groups of 128 programs. For example, let's say that a device has 512 Programs. It may divide these into 4 banks of 128 programs apiece. So, if you want program #129, that would actually be the first program within the second bank. You would send a Bank Select Controller to switch to the second bank (ie, the first bank is #0), and then follow with a Program Change to select the first Program in this bank. If a MultiTimbral device, then each Part usually can be set to its own Bank/Program.

On MultiTimbral devices that have a Drum Part, the Bank Select is sometimes used to switch between "Drum Kits".

NOTE: When a Bank Select is received, the MIDI module doesn't actually change to a patch in the new bank. Rather, the Bank Select value is simply stored by the MIDI module without changing the current patch. Whenever a subsequent Program Change is received, the stored Bank Select is **then** utilized to switch to the specified patch in the new bank. For this reason, Bank Select must be sent **before** a Program Change, when you desire changing to a patch in a different bank. (Of course, if you simply wish to change to another patch in the same bank, there is no need to send a Bank Select first).

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF.

NOTE: Most devices use the Coarse adjust (#0) alone to switch banks since most devices don't have more than 128 banks (of 128 Patches each).

MOD Wheel

Number: 1 (coarse) 33 (fine)

Affects:

Sets the *MOD Wheel* to a particular value. Usually, MOD Wheel introduces some sort of (LFO) vibrato effect. If a MultiTimbral device, then each Part usually has its own MOD Wheel setting.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is no modulation effect.

Breath Controller

Number: 2 (coarse) 34 (fine)

Affects:

Whatever the musician sets this controller to affect. Often, this is used to control a parameter such as what Aftertouch can. After all, breath control is a wind player's version of how to vary pressure. If a MultiTimbral device, then each Part usually has its own Breath Controller setting.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is minimum breath pressure.

Foot Pedal

Number: 4 (coarse) 36 (fine)

Affects:

Whatever the musician sets this controller to affect. Often, this is used to control the a parameter such as what Aftertouch can. This foot pedal is a continuous controller (ie, potentiometer). If a MultiTimbral device, then each Part usually has its own Foot Pedal value.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is minimum effect.

Portamento Time

Number: 5 (coarse) 37 (fine)

Affects:

The rate at which portamento slides the pitch between 2 notes. If a MultiTimbral device, then each Part usually has its own Portamento Time.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is slowest rate.

Data Entry Slider

Number: 6 (coarse) 38 (fine)

The value of some Registered or Non-Registered Parameter. Which parameter is affected depends upon a preceding RPN or NRPN message (which itself identifies the parameter's number).

On some devices, this slider may not be used in conjunction with RPN or NRPN messages. Instead the musician can set the slider to control a single parameter directly, often a parameter such as what Aftertouch can control.

If a MultiTimbral device, then each Part usually has its own RPN and NRPN settings, and Data Entry slider setting.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is minimum effect.

Volume

Number: 7 (coarse) 39 (fine)

Affects:

The device's volume level. If a MultiTimbral device, then each Part has its own volume. In this case, a device's master volume may be controlled by another method such as the Univeral SysEx Master Volume message, or take its volume from one of the Parts, or be controlled by a General Purpose Slider controller.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is no volume at all.

NOTE: Most all devices ignore the Fine adjust (#39) for Volume, and just implement Coarse adjust (#7) because 14-bit resolution isn't needed for this. In this case, maximum is 127 and off is 0.

Balance

Number: 8 (coarse) 40 (fine)

Affects:

The device's stereo balance (assuming that the device has stereo audio outputs). If a MultiTimbral device, then each Part usually has its own Balance. This is generally when Balance becomes useful, because then you can use Pan, Volume, and Balance controllers to internally mix all of the Parts to the device's stereo outputs. Typically, Balance would be used on a Part that had stereo elements (where you wish to adjust the volume of the stereo elements without changing their pan positions), whereas Pan is more appropriate for a Part that is strictly a "mono instrument".

Value Range:

14-bit coarse/fine resolution. 16,384 possible setting, 0x0000 to 0x3FFF where 0x2000 is center balance, 0x0000 emphasizes the left elements mostly, and 0x3FFF emphasizes the right elements mostly. Some devices only respond to coarse adjust (128 settings) where 64 is center, 0 is leftmost emphsis, and 127 is rightmost emphasis.

NOTE: Most all devices ignore the Fine adjust (#40) for Balance, and just implement Coarse adjust (#8) because 14-bit resolution isn't needed for this.

Pan

Number: 10 (coarse) 42 (fine)

Affects:

Where within the stereo field the device's sound will be placed (assuming that it has stereo audio outputs). If a MultiTimbral device, then each Part usually has its own pan position. This is generally when Pan becomes useful, because then you can use Pan, Volume, and Balance controllers to internally mix all of the Parts to the device's stereo outputs.

Value Range:

14-bit coarse/fine resolution. 16,384 possible positions, 0x0000 to 0x3FFF where 0x2000 is center position, 0x0000 is hard left, and 0x3FFF is hard right. Some devices only respond to coarse adjust (128 positions) where 64 is center, 0 is hard left, and 127 is hard right.

NOTE: Most all devices ignore the Fine adjust (#42) for Pan, and just implement Coarse adjust (#10) because 14-bit resolution isn't needed for this.

Expression

Number: 11 (coarse) 43 (fine)

Affects:

This is a percentage of Volume (ie, as set by Volume Controller). In other words, Expression divides the current volume into 16,384 steps (or 128 if 8-bit instead of 14-bit resolution is used). Volume Controller is used to set the overall volume of the entire musical part, whereas Expression is used for doing crescendos and decrescendos. By having both a master Volume and sub-Volume (ie, Expression), it makes possible adjusting the overall volume of a part without having to adjust every single MIDI message comprising a crescendo or decrescendo. When Expression is at 100% (ie, maximum or 0x3FFF), then the volume represents the true setting of Volume Controller. Lower values of Expression begin to subtract from the volume. When Expression is 0% (ie, 0x0000), then volume is off. When Expression is 50% (ie, 0x1FFF), then the volume is cut in half.

If a MultiTimbral device, then each Part usually has its own Expression level.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is minimum effect.

NOTE: Most all devices ignore the Fine adjust (#43) for Expression, and just implement Coarse adjust (#11) because 14-bit resolution isn't needed for this. In this case, maximum is 127, 50% is 64, and off is 0. So assume that a channel's volume is 100. After receiving an Expression Controller of value 64, the volume is reduced to 50 (ie, 50% of 100). After receiving another Expression of 127, the volume is restored to 100. Now, assume the volume is changed (via Volume Controller) to 80. After receiving an Expression Controller of value 64, the volume is reduced to 40 (ie, 50% of 80). After receiving another Expression of 127, the volume is restored to 80.

Effect Control 1

Number: 12 (coarse) 44 (fine)

Affects:

This can control any parameter relating to an effects device, such as the Reverb Decay Time for a reverb unit built into a GM sound module.

NOTE: There are separate controllers for setting the Levels (ie, volumes) of Reverb, Chorus, Phase Shift, and other effects.

If a MultiTimbral device, then each Part usually has its own Effect Control 1.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is minimum effect.

Effect Control 2

Number: 13 (coarse) 45 (fine)

Affects:

This can control any parameter relating to an effects device, such as the Reverb Decay Time for a reverb unit built into a GM sound module.

NOTE: There are separate controllers for setting the Levels (ie, volumes) of Reverb, Chorus, Phase Shift, and other effects.

If a MultiTimbral device, then each Part usually has its own Effect Control 2.

Value Range:

14-bit coarse/fine resolution. 0x0000 to 0x3FFF where 0 is minimum effect.

General Purpose Slider

Number: 16, 17, 18, 19

Affects:

Whatever the musician sets this controller to affect. There are 4 General Purpose Sliders, with the above controller numbers. Often, these are used to control parameters such as what Aftertouch can. If a MultiTimbral device, then each Part usually has its own responses to the 4 General Purpose Sliders. Note that these sliders don't have a fine adjustment.

Value Range:

0x00 to 0x7F where 0 is minimum effect.

Hold Pedal

Number: 64

Affects:

When on, this holds (ie, sustains) notes that are playing, even if the musician releases the notes (ie, the Note Off effect is postponed until the musician switches the Hold Pedal off). If a MultiTimbral device, then each Part usually has its own Hold Pedal setting.

NOTE: When on, this also postpones any All Notes Off controller message on the same channel.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

Portamento

Number: 65

Affects:

Whether the portamento effect is on or off. If a MultiTimbral device, then each Part usually has its own portamento on/off setting.

NOTE: There is another controller to set the portamento time.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

Sustenuto

Number: 66

Affects:

Like the Hold Pedal controller, except this only sustains notes that are already on (ie, the device has received Note On messages, but the respective Note Offs haven't yet arrived) *when the pedal is turned on*. After the pedal is on, it continues to hold these initial notes all of the while that the pedal is on, but during that time, all other arriving Note Ons are not held. So, this pedal implements a "chord hold" for the notes that are sounding when this pedal is turned on. If a MultiTimbral device, then each Part usually has its own Sustenuto setting.

NOTE: When on, this also postpones any All Notes Off controller message on the same channel for those notes being held.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

Soft Pedal

Number: 67

Affects:

When on, this lowers the volume of any notes played. If a MultiTimbral device, then each Part usually has its own Soft Pedal setting.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

Legato Pedal

Number: 68

Affects:

When on, this causes a legato effect between notes, which is usually achieved by skipping the attack portion of the VCA's envelope. Use of this controller allows a keyboard player to better simulate the phrasing of wind and brass players, who often play several notes with a single tonguing, or simulate guitar pull-offs and hammer-ons (ie, where secondary notes are not picked). If a MultiTimbral device, then each Part usually has its own Legato Pedal setting.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

Hold 2 Pedal

Number: 69

Affects:

When on, this lengthens the release time of the playing notes' VCA (ie, makes the note take longer to fade out after it's released, versus when this pedal is off). Unlike the other Hold Pedal controller, this pedal doesn't permanently sustain the note's sound until the musician releases the pedal. Even though the note takes longer to fade out when this pedal is on, the note may eventually fade out despite the musician still holding down the key *and* this pedal. If a MultiTimbral device, then each Part usually has its own Hold 2 Pedal setting.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

Sound Variation

Number: 70

Affects:

Any parameter associated with the circuitry that produces sound. For example, if a device uses looped digital waveforms to create sound, this controller may adjust the sample rate (ie, playback speed), for a "tuning" control. If a MultiTimbral device, then each Part usually has its own patch with its respective VCA, VCF, tuning, sound sources, etc, parameters that can be adjusted with this controller.

NOTE: There are other controllers for adjusting VCA attack and release times, VCF cutoff frequency, and other generic sound parameters.

Value Range:

0 to 127, with 0 being minimum setting.

Sound Timbre

Number: 71

Affects:

Controls the (VCF) filter's envelope levels, for a "brightness" control. If a MultiTimbral device, then each Part usually has its own patch with its respective VCF cutoff frequency that can be adjusted with this controller.

NOTE: There are other controllers for adjusting VCA attack and release times, and other generic sound parameters.

Value Range:

0 to 127, with 0 being minimum setting.

Sound Release Time

Number: 72

Affects:

Controls the (VCA) amp's envelope release time, for a control over how long it takes a sound to fade out. If a MultiTimbral device, then each Part usually has its own patch with its respective VCA envelope that can be adjusted with this controller.

NOTE: There are other controllers for adjusting VCA attack time, VCF cutoff frequency, and other generic sound parameters.

Value Range:

0 to 127, with 0 being minimum setting.

Sound Attack Time

Number: 73

Affects:

Controls the (VCA) amp's envelope attack time, for a control over how long it takes a sound to fade in. If a MultiTimbral device, then each Part usually has its own patch with its respective VCA envelope that can be adjusted with this controller.

NOTE: There are other controllers for adjusting VCA release time, VCF cutoff frequency, and other generic sound parameters.

Value Range:

0 to 127, with 0 being minimum setting.

Sound Brightness

Number: 74

Affects:

Controls the (VCF) filter's cutoff frequency, for a "brightness" control. If a MultiTimbral device, then each Part usually has its own patch with its respective VCF cutoff frequency that can be adjusted with this controller.

NOTE: There are other controllers for adjusting VCA attack and release times, and other generic sound parameters.

Value Range:

0 to 127, with 0 being minimum setting.

Sound Control 6, 7, 8, 9, 10

Number: 75, 76, 77, 78, 79

Affects:

These 5 controllers can be used to adjust any parameters associated with the circuitry that produces sound. For example, if a device uses looped digital waveforms to create sound, one controller may adjust the sample rate (ie, playback speed), for a "tuning" control. If a MultiTimbral device, then each Part usually has its own patch with its respective VCA, VCF, tuning, sound sources, etc, parameters that can be adjusted with these controllers.

NOTE: There are other controllers for adjusting VCA attack and release times, and VCF cutoff frequency.

Value Range:

0 to 127, with 0 being minimum setting.

General Purpose Button

Number: 80, 81, 82, 83

Affects:

Whatever the musician sets this controller to affect. There are 4 General Purpose Buttons, with the above controller numbers. This are either on or off, so they are often used to implement on/off functions, such as a Metronome on/off switch on a sequencer. If a MultiTimbral device, then each Part usually has its own responses to the 4 General Purpose Buttons.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

Effects Level

Number: 91

Affects:

The effects amount (ie, level) for the device. Often, this is the reverb or delay level. If a MultiTimbral device, then each Part usually has its own effects level.

Value Range:

0 to 127, with 0 being no effect applied at all.

Tremulo Level

Number: 92

Affects:

The tremulo amount (ie, level) for the device. If a MultiTimbral device, then each Part Parts usually has its own tremulo level.

Value Range:

0 to 127, with 0 being no tremulo applied at all.

Chorus Level

Number: 93

Affects:

The chorus effect amount (ie, level) for the device. If a MultiTimbral device, then each Part usually has its own chorus level.

Value Range:

0 to 127, with 0 being no chorus effect applied at all.

Celeste Level

Number: 94

Affects:

The celeste (detune) amount (ie, level) for the device. If a MultiTimbral device, then each Part usually has its own celeste level.

Value Range:

0 to 127, with 0 being no celeste effect applied at all.

Phaser Level

Number: 95

Affects:

The Phaser effect amount (ie, level) for the device. If a MultiTimbral device, then each Part usually has its own Phaser level.

Value Range:

0 to 127, with 0 being no phaser effect applied at all.

Data Button increment

Number: 96

Affects:

Causes a Data Button to increment (ie, increase by 1) its current value. Usually, this data button's value is being used to set some Registered or Non-Registered Parameter. Which RPN or NRPN parameter is being affected depends upon a preceding RPN or NRPN message (which itself identifies the parameter's number).

Value Range:

The value byte isn't used and defaults to 0.

Data Button decrement

Number: 97

Affects:

Causes a Data Buttonto decrement (ie, decrease by 1) its current value. Usually, this data button's value is being used to set some Registered or Non-Registered Parameter. Which RPN or NRPN parameter is being affected depends upon a preceding RPN or NRPN message (which itself identifies the parameter's number).

Value Range:

The value byte isn't used and defaults to 0.

Registered Parameter Number (RPN)

Number: 101 (coarse) 100 (fine)

Affects:

Which parameter the Data Button Increment, Data Button Decrement, or Data Entry controllers affect. Since RPN has a coarse/fine pair (14-bit), the number of parameters that can be registered is 16,384. That's a lot of parameters that a MIDI device could allow to be controlled over MIDI. It's up to the IMA to assign Registered Parameter Numbers to specific functions.

Value Range:

0 to 16,384 where each value stands for a different RPN. Here are the currently registered parameter numbers:

Pitch Bend Range (ie, Sensitivity) 0x0000

NOTE: The coarse adjustment (usually set via Data Entry 6) sets the range in semitones. The fine adjustment (usually set via Data Entry 38) set the range in cents. For example, to adjust the pitch wheel range to go up/down 2 semitones and 4 cents:

```
B0 65 00  Controller/chan 0, RPN coarse (101), Pitch Bend Range

B0 64 00  Controller/chan 0, RPN fine (100), Pitch Bend Range

B0 06 02  Controller/chan 0, Data Entry coarse, +/- 2 semitones

B0 26 04  Controller/chan 0, Data Entry fine, +/- 4 cents
```

Master Fine Tuning (ie, in cents) 0x0001

NOTE: Both the coarse and fine adjustments together form a 14-bit value that sets the tuning in semitones, where 0x2000 is A440 tuning.

Master Coarse Tuning (in half-steps) 0x0002

NOTE: Setting the coarse adjustment adjusts the tuning in semitones, where 0x40 is A440 tuning. There is no need to set a fine adjustment.

RPN Reset 0x3FFF

NOTE: No coarse or fine adjustments are applicable. This is a "dummy" parameter.

Here's the way that you use RPN. First, you decide which RPN you wish to control. Let's say that we wish to set Master Fine Tuning on a device. That's RPN 0x0001. We need to send the device the RPN Coarse and RPN Fine controller messages in order to tell it to affect RPN 0x0001. So, we divide the 0x0001 into 2 bytes, the fine byte and the coarse byte. The fine byte contains bits 0 to 6 of the 14-bit value. The coarse byte contains bits 7 to 13 of the 14-bit value, right-justified. So, here are the RPN Coarse and Fine messages (assuming that the device is responding to MIDI channel 0):

```
B0 65 00  Controller/chan 0, RPN coarse (101), bits

            7 to 13 of 0x0001, right-justified (with high bit clear)

B0 64 01  Controller/chan 0, RPN fine (100), bits

            0 to 6 of 0x0001, (with high bit clear)
```

Now, we've just told the device that any Data Button Increment, Data Button decrement, or Data Entry controllers it receives should affect the Master Fine Tuning. Let's say that we wish to set this tuning to the 14-bit value 0x2000 (which happens to be centered tuning). We could use the Data Entry (coarse and fine) controller messages as so to send that 0x2000:

```
B0 06 40  Controller/chan 0, Data Entry coarse (6), bits

            7 to 13 of 0x2000, right-justified (with high bit clear)

B0 26 00  Controller/chan 0, Data Entry fine (38), bits

            0 to 6 of 0x2000, (with high bit clear)
```

As a final example, let's say that we wish to increment the Master Fine Tuning by one (ie, to 0x2001). We could use the Data Entry messages again. Or, we could use the Data Button Increment, which doesn't have a coarse/fine pair of controller numbers like Data Entry.

```
B0 60 00  Controller/chan 0, Data Button Increment (96),

               last byte is unused
```

Of course, if the device receives RPN messages for another parameter, then the Data Button Increment, Data Button Decrement, and Data Entry controllers will switch to adjusting that parameter.

RPN 0x3FFF (reset) forces the Data Button increment, Data Button decrement, and Data Entry controllers to not adjust any RPN (ie, disables those buttons' adjustment of any RPN).

Non-Registered Parameter Number (NRPN)

Number: 99 (coarse) 98 (fine)

Affects:

Which parameter the Data Button Increment, Data Button Decrement, or Data Entry controllers affect. Since NRPN has a coarse/fine pair (14-bit), the number of parameters that can be registered is 16,384. That's a lot of parameters that a MIDI device could allow to be controlled over MIDI. It's entirely up to each manufacturer which parameter numbers are used for whatever purposes. These don't have to be registered with the IMA.

Value Range:

The same scheme is used as per the Registered Parameter controller. Refer to that. By contrast, the coarse/fine messages for NRPN for the preceding RPN example would be:

```
B0 63 00

B0 62 01
```

NOTE: Since each device can define a particular NRPN controller number to control anything, it's possible that 2 devices may interpret the same NRPN number in different manners. Therefore, a device should allow a musician to disable receipt of NRPN, in the event that there is a conflict between the NRPN implementations of 2 daisy-chained devices.

All Controllers Off

Number: 121

Affects:

Resets all controllers to default states. This means that all switches (such as Hold Pedal) are turned off, and all continuous controllers (such as Mod Wheel) are set to minimum positions. If the device is MultiTimbral, this only affects the Part assigned to the MIDI channel upon which this message is received.

Value Range:

The value byte isn't used and defaults to 0.

Local Keyboard on/off

Number: 122

Affects:

Turns the device's keyboard on or off locally. If off, the keyboard is disconnected from the device's internal sound generation circuitry. So when the musician presses keys, the device doesn't trigger any of its internal sounds. But, the keyboard still generates Note On, Note Off, Aftertouch, and Channel Pressure messages. In this way, a musician can eliminate a situation where MIDI messages get looped back (over MIDI cables) to the device that created those messages. Furthermore, if a device is only going to be played remotely via MIDI, then the keyboard may be turned off in order to allow the device to concentrate more on dealing with MIDI messages rather than scanning the keyboard for depressed notes and varying pressure.

Value Range:

0 (to 63) is off. 127 (to 64) is on.

All Notes Off

Number: 123

Affects:

Turns off all notes that were turned on by received Note On messages, and which haven't yet been turned off by respective Note Off messages. This message is not supposed to turn off any notes that the musician is playing on the local keyboard. So, if a device can't distinguish between notes played via its *MIDI IN* and notes played on the local keyboard, it should not implement All Notes Off. Furthermore, if a device is in Omni On state, it should ignore this message on *any* channel.

NOTE: If the device's Hold Pedal controller is on, the notes aren't actually released until the Hold Pedal is turned off. See All Sound Off controller message for turning off the sound of these notes immediately.

Value Range:

The value byte isn't used and defaults to 0.

All Sound Off

Number: 120

Affects:

Mutes all sounding notes that were turned on by received Note On messages, and which haven't yet been turned off by respective Note Off messages. This message is not supposed to mute any notes that the musician is playing on the local keyboard. So, if a device can't distinguish between notes played via its *MIDI IN* and notes played on the local keyboard, it should not implement All Sound Off.

NOTE: The difference between this message and All Notes Off is that this message immediately mutes all sound on the device regardless of whether the Hold Pedal is on, and mutes the sound quickly regardless of any lengthy VCA release times. It's often used by sequencers to quickly mute all sound when the musician presses "Stop" in the middle of a song.

Value Range:

The value byte isn't used and defaults to 0.

Omni Off

Number: 124

Affects:

Turns *Omni* off. See the discussion on MIDI Modes.

Value Range:

The value byte isn't used and defaults to 0.

NOTE: When a device receives an Omni Off message, it should automatically turn off all playing notes.

Omni On

Number: 125

Affects:

Turns *Omni* on. See the discussion on MIDI Modes.

Value Range:

The value byte isn't used and defaults to 0.

NOTE: When a device receives an Omni On message, it should automatically turn off all playing notes.

Monophonic Operation

Number: 126

Affects:

Enables *Monophonic operation* (thus disabling Polyphonic operation). See the discussion on MIDI Modes.

Value Range:

If Omni is off, this Value tells how many MIDI channels the device is expected to respond to in Mono mode. In other words, if Omni is off, this value is used to select a limited set of the 16 MIDI channels (ie, 1 to 16) to respond to.Conversely, if Omni is on, this Value is ignored completely, and the device only ever plays one note at a time (unless a MultiTimbral device). So, the following discussion is only relevant if Omni Off.

If Value is 0, then the number of MIDI channels that the device will respond to simultaneously will be equal to how many voices the device can sound simultaneously. In other words, if the device can sound at least 16 voices simultaneously, then it can respond to Voice Category messages on all 16 channels. Of course, being Monophonic operation, the device can only sound one note at a time per each MIDI channel. So, it can sound a note on channel 1 and channel 2 simultaneously, for example, but can't sound 2 notes both on channel 1 simultaneously.

Of course, MultiTimbral devices completely violate the preceding theory. MultiTimbral devices always can play polyphonically on each MIDI channel. If Value is 0, what this means is that the device can play as many MIDI channels as it has Parts. So, if the device can play 16 of its patches simultaneously, then it can respond to Voice Category messages on all 16 channels.

If Value is not 0 (ie, 1 to 16), then that's how many MIDI channels the device is allowed to respond to. For example, a value of 1 would mean that the device would only be able to respond to 1 MIDI channel. Since the device is also limited to sounding only 1 note at a time on that MIDI channel, then the device would truly be a Monophonic instrument incapable of sounding more than one note at a time. If a device is asked to respond to more MIDI channels than it has voices to accomodate, then it will handle only as many MIDI channels as it has voices. For example, if an 8-voice synth, on Base Channel 0, receives the value 16 in the Mono message, then the synth will play messages on MIDI channels 0 to 7 and ignore messages on 8 to 15.

Again, MultiTimbral devices violate the above theory. A value of 1 would mean that the device would only be able to respond to 1 MIDI channel (and therefore only play 1 Part), but would do so Polyphonically. If a MultiTimbral device is asked to respond to more MIDI channels than it has Parts to accomodate, then it will handle only as many MIDI channels as it has Parts. For example, if a device can play only 5 Patches simultaneously, and receives the value 8 in the Mono message, then the device will play 5 patches on MIDI channels 0 to 4 and ignore messages on channels 5 to 7.

Most devices capable of Monophonic operation, allow the user to specify a *Base Channel*. This will be the lowest MIDI channel that the device responds to. For example, if a Mono message specifies that the device is to respond to only 2 channels, and its Base Channel is 2, then the device responds to channels 2 and 3.

NOTE: When a device receives a Mono Operation message, it should automatically turn off all playing notes.

Polyphonic Operation

Number: 127

Affects:

Enables *Polyphonic operation* (thus disabling Monophonic operation). See the discussion on MIDI Modes.

Value Range:

The value byte isn't used and defaults to 0.

NOTE: When a device receives a Poly Operation message, it should automatically turn off all playing notes.

MIDI Modes

Some MIDI devices can be switched in and out of *Omni* state.

When Omni is off, a MIDI device can only respond to Voice Category messages (ie, Status bytes of 0x80 to 0xEF) upon a limited number of channels, usually only 1. Typically, the device allows the musician to pick one of the 16 MIDI channels that the device will respond to. This is then referred to as the device's *Base Channel*. So for example, if a device's Base Channel is set to 1, and a Voice Category message upon channel 2 arrives at the device's MIDI IN, the device ignores that message.

NOTE: Virtually all modern devices allow the musician to manually choose the Base Channel. A device may even define its own SysEx message that can change its Base Channel. Remember that SysEx messages are of the System Common Category, and therefore aren't (normally) tied to the Base Channel itself.

When Omni is on, a device doesn't respond to just one MIDI channel, but rather, responds to all 16 MIDI channels. The only benefit of Omni On is that, regardless of which channel any message is received upon, a device always responds to the message. This mades it very foolproof for a musician to hook up two devices and always have one device respond to the other regardless of any MIDI channel discrepancies between the device generating the data (ie, referred to as the *transmitter*) and the device receiving the data (ie, referred to as the *receiver*). Of course, if the musician daisy-chains another device, and he wants the 2 devices to play different musical parts, then he has to switch Omni Off on both devices. Otherwise, a device with Omni On will respond to messages intended for the other device (as well as messages intended for itself).

NOTE: Omni can be switched on or off with the Omni On and Omni Off controller messages. But these messages must be received upon the device's Base Channel in order for the device to respond to them. What this implies is that even when a device is in Omni On state (ie, capable of responding to all 16 channels), it still has a Base Channel for the purpose of turning Omni On or Off.

One might think that MultiTimbral devices employ Omni On. Because you typically may choose (upto) 16 different Patches, each playing its own musical part, you need the device to be able to respond to more than one MIDI channel so that you can assign each Patch to a different MIDI channel. Actually, MultiTimbral devices do not use Omni On for this purpose. Rather, the device regards itself as having 16 separate sound modules (ie, Parts) inside of it, with each module in Omni Off mode, and capable of being set to its own Base Channel. Usually, you also have a "master" Base Channel which may end up having to be set the same as one of the individual Parts. Most MultiTimbral devices offer the musician the choice of which particular channels to use, and which to ignore (if he doesn't need all 16 patches playing simultaneously on different channels). In this way, he can daisy-chain another multitimbral device and use any ignored channels (on the first device) with this second device. Unfortunately, the MIDI spec has no specific "MultiTimbral" mode message. So, a little "creative reinterpretation" of Monophonic mode is employed, as you'll learn in a moment.

In addition to Omni On or Off, many devices can be switched between Polyphonic or Monophonic operation.

In Polyphonic operation, a device can respond to more than one Note On upon a given channel. In other words, it can play chords on that channel. For example, assume that a device is responding to Voice Category messages on channel 1. If the device receives a Note On for middle C on channel 1, it will sound that note. If the device then receives a Note On for high C also on channel 1 (before receiving a Note Off for middle C), the device will sound the high C as well. Both notes will then be sounding simultaneously.

In Monophonic operation, a device can only respond to one Note On at a time upon a given channel. It can't play chords; only single note "melodies". For example, assume that a device is responding to Voice Category messages on channel 1. If the device receives a Note On for middle C on channel 1, it will play that note. If the device then receives a Note On for high C (before receiving a Note Off for middle C), the device will automatically turn off the middle C before playing the high C. So what's the use of forcing a device capable of playing chords into such a Monophonic state? Well, there are lots of Monophonic instruments in the world, for example, most brass and woodwinds. They can only play one note at a time. If using a Trumpet Patch, a keyboard player might want to force a device into Monophonic operation in order to better simulate a Trumpet. Some devices have special effects that only work in Monophonic operation such as Portamento, and smooth transition between notes (ie, skipping the VCA attack when moving from one Note On that "overlaps" another Note On -- this is often referred to as *legato* and makes for a more realistic musical performance for brass and woodwind patches). That's in theory how Mono operation is supposed to work, but MultiTimbral devices created long after the MIDI spec was designed, had to subvert Mono operation into Polyphonic operation in order to come up with a "MultiTimbral mode", as you'll learn.

NOTE: A device can be switched between Polyphonic or Monophonic with the Polyphonic and Monophonic controller messages. But these messages must be received upon the device's Base Channel in order for the device to respond to them.

Of course, a MIDI device could have Omni On and be in Polyphonic state. Or, the device could have Omni On but be in Monophonic state. Or, the device could have Omni Off and be in Polyphonic state. Or, the device could have Omni Off but be in Monophonic state. There are 4 possible combinations here, and MIDI refers to these as 4 *Modes*. For example, Mode 1 is the aforementioned Omni On / Polyphonic state. Here are the 4 Modes:

Mode 1 - Omni On / Poly

The device plays *all* MIDI data received on all 16 MIDI channels. If a MultiTimbral device, then it often requires the musician to manually select which one Patch to play all 16 channels, and this setting is usually saved in "patch memory".

Mode 2 - Omni On / Mono

The device plays only one note out of all of the MIDI data received on all 16 MIDI channels. This mode is seldom implemented because playing one note out of all the data happening on all 16 channels is not very useful.

Mode 3 - Omni Off / Poly

The device plays all MIDI data received on 1 specific MIDI channel. The musician usually gets to choose which channel he wants that to be. If a MultiTimbral device, then it often requires the musician to manually select which one Patch to play that MIDI channel, and this setting is usually saved in "patch memory".

Mode 4 - Omni Off / Mono

In theory, the device plays one note at a time on 1 (or more) specific MIDI channels. In practice, the manufacturers of MultiTimbral threw the entire concept of Monophonic out the window, and use this for "MultiTimbral mode". On a MultiTimbral device, this mode means that the device plays polyphonically on 1 (or more) specific MIDI channels. The Monophonic controller message has a Value associated with it. This Value is applicable in Mode 4 (whereas it's ignored in Mode 2), and determines how many MIDI channels are responded to. If 1, then on a non-MultiTimbral device, this would give you a truly monophonic instrument. Of course, on a MultiTimbral device, it gives you the same thing as Mode 3. If the Value is 0, then a non-MultiTimbral device uses as many MIDI channels as it has voices. So, for an 8 voice synth, it would use 8 MIDI Channels, and each of those channels would play one note at a time. For a MultiTimbral device, if the Value is 0, then the device uses as many MIDI channels as it has Parts. So, if a MultiTimbral device can play only 8 patches simultaneously, then it would use 8 MIDI Channels, and each of those channels could play polyphonically.

Some devices do not support all of these modes. The device should ignore controller messages which attempt to switch it into an unsupported state, or switch to the next closest mode.

If a device doesn't have some way of saving the musician's choice of Mode when the unit is turned off, the device should default to Mode 1 upon the next power up.

On final question arises. If a MultiTimbral device doesn't implement a true monophonic mode for Mode 4, then how do you get one of its Parts to play in that useful Monophonic state (ie, where you have Portamento and legato features)? Well, many MultiTimbral devices allow a musician to manually enable a "Solo Mode" per each Part. Some devices even use the *Legato Pedal controller* (or a *General Purpose Button controller*) to enable/disable that function, so that you can turn it on/off for each Part over MIDI.

NOTE: A device that can both generate MIDI messages (ie, perhaps from an electronic piano keyboard) as well as receive MIDI messages (ie, to be played on its internal sound circuitry), is allowed to have its transmitter set to a different Mode and MIDI channel than its receiver, if this is desired. In fact, on MultiTimbral devices with a keyboard, the keyboard often has to switch between MIDI channels so that the musician can access the Parts one at a time, without upsetting the MIDI channel assignments for those Parts.

RealTime Category Messages

Each RealTime Category message (ie, Status of 0xF8 to 0xFF) consists of only 1 byte, the Status. These messages are primarily concerned with timing/syncing functions which means that they must be sent and received at specific times without any delays. Because of this, MIDI allows a RealTime message to be sent **at any time**, even interspersed within some other MIDI message. For example, a RealTime message could be sent inbetween the two data bytes of a Note On message. A device should always be prepared to handle such a situation; processing the 1 byte RealTime message, and then subsequently resume processing the previously interrupted message as if the RealTime message had never occurred.

For more information about RealTime, read the sections Running Status, Ignoring MIDI Messages, and Syncing Sequence Playback.

Running Status

The MIDI spec allows for a MIDI message to be sent without its Status byte (ie, just its data bytes are sent) **as long as the previous, transmitted message had the same Status**. This is referred to as *running status*. Running status is simply a clever scheme to maximize the efficiency of MIDI transmission (by removing extraneous Status bytes). The basic philosophy of running status is that a device must always remember the last Status byte that it received (except for RealTime), and if it doesn't receive a Status byte when expected (on subsequent messages), it should assume that it's dealing with a running status situation. A device that generates MIDI messages should always remember the last Status byte that it sent (except for RealTime), and if it needs to send another message with the same Status, the Status byte may be omitted.

Let's take an example of a device creating a stream of MIDI messages. Assume that the device needs to send 3 Note On messages (for middle C, E above middle C, and G above middle C) on channel 0. Here are the 3 MIDI messages to which I'm referring.

```
0x90 0x3C 0x7F
```

```
0x90 0x40 0x7F

0x90 0x43 0x7F
```

Notice that the Status bytes of all 3 messages are the same (ie, Note On, Channel 0). Therefore the device could implement running status for the latter 2 messages, sending the following bytes:

```
0x90 0x3C 0x7F

0x40 0x7F

0x43 0x7F
```

This allows the device to save time since there are 2 less bytes to transmit. Indeed, if the message that the device sent before these 3 also happened to be a Note On message on channel 0, then the device could have omitted the first message's Status too.

Now let's take the perspective of a device receiving this above stream. It receives the first message's Status (ie, 0x90) and thinks "Here's a Note On Status on channel 0. I'll remember this Status byte. I know that there are 2 more data bytes in a Note On message. I'll expect those next". And, it receives those 2 data bytes. Then, it receives the data byte of the second message (ie, 0x40). Here's when the device thinks "I didn't expect another data byte. I expected the Status byte of some message. This must be a running status message. The last Status byte that I received was 0x90, so I'll assume that this is the same Status. Therefore, this 0x40 is the first data byte of another Note On message on channel 0".

Remember that a Note On message with a velocity of 0 is really considered to be a Note Off. With this in mind, you could send a whole stream of note messages (ie, turning notes on and off) without needing a Status byte for all but the first message. All of the messages will be Note On status, but the messages that really turn notes off will have 0 velocity. For example, here's how to play and release middle C utilizing running status:

```
0x90 0x3C 0x7F

0x3C 0x00     <-- This is really a Note Off because of 0 velocity
```

RealTime Category messages (ie, Status of 0xF8 to 0xFF) do not effect running status in any way. Because a RealTime message consists of only 1 byte, and it may be received at any time, including interspersed with another message, it should be handled transparently. For example, if a 0xF8 byte was received inbetween any 2 bytes of the above examples, the 0xF8 should be processed immediately, and then the device should resume processing the example streams exactly as it would have otherwise. Because RealTime messages only consist of a Status, running status obviously can't be implemented on RealTime messages.

System Common Category messages (ie, Status of 0xF0 to 0xF7) cancel any running status. In other words, the message after a System Common message **must** begin with a Status byte. System Common messages themselves can't be implemented with running status. For example, if a Song Select message was sent immediately after another Song Select, the second message would still need a Status byte.

Running status is only implemented for Voice Category messages (ie, Status is 0x80 to 0xEF).

A recommended approach for a receiving device is to maintain its "running status buffer" as so:

1. Buffer is cleared (ie, set to 0) at power up.
2. Buffer stores the status when a Voice Category Status (ie, 0x80 to 0xEF) is received.
3. Buffer is cleared when a System Common Category Status (ie, 0xF0 to 0xF7) is received.
4. Nothing is done to the buffer when a RealTime Category message is received.
5. Any data bytes are ignored when the buffer is 0.

Syncing Sequence Playback

A sequencer is a software program or hardware unit that "plays" a musical performance complete with appropriate rhythmic and melodic inflections (ie, plays musical notes in the context of a musical beat).

Often, it's necessary to synchronize a sequencer to some other device that is controlling a timed playback, such as a drum box playing its internal rhythm patterns, so that both play at the same instant and the same tempo. Several MIDI messages are used to cue devices to start playback at a certain point in the sequence, make sure that the devices start simultaneously, and then keep the devices in sync until they are simultaneously stopped. One device, the master, sends these messages to the other device, the slave.The slave references its playback to these messages.

The message that controls the playback rate (ie, ultimately tempo) is MIDI Clock. This is sent by the master at a rate dependent upon the master's tempo. Specifically, the master sends 24 MIDI Clocks, spaced at equal intervals, during every quarter note interval.(12 MIDI Clocks are in an eighth note, 6 MIDI Clocks in a 16th, etc). Therefore, when a slave device counts down the receipt of 24 MIDI Clock messages, it knows that one quarter note has passed. When the slave counts off another 24 MIDI Clock messages, it knows that another quarter note has passed.

For example, if a master is set at a tempo of 120 BPM (ie, there are 120 quarter notes in every minute), the master sends a MIDI clock every 20833 microseconds. (ie, There are 1,000,000 microseconds in a second. Therefore, there are 60,000,000 microseconds in a minute. At a tempo of 120 BPM, there are 120 quarter notes per minute. There are 24 MIDI clocks in each quarter note. Therefore, there should be 24 * 120 MIDI Clocks per minute. So, each MIDI Clock is sent at a rate of 60,000,000/(24 * 120) microseconds).

The master needs to be able to start the slave precisely when the master starts. The master does this by sending a MIDI Start message. The MIDI Start message alerts the slave that, upon receipt of the very next MIDI Clock message, the slave should start the playback of its sequence. In other words, the MIDI Start puts the slave in "play mode", and the receipt of that first MIDI Clock marks the initial downbeat of the song (ie, *MIDI Beat* 0). What this means is that (typically) the master sends out that MIDI Clock "downbeat" **immediately** after the MIDI Start. (In practice, most masters allow a 1 millisecond interval inbetween the MIDI Start and subsequent MIDI Clock messages in order to give the slave an opportunity to prepare itself for playback). In essense, a MIDI Start is just a warning to let the slave know that the next MIDI Clock represents the downbeat, and playback is to start then. Of course, the slave then begins counting off subsequent MIDI Clock messages, with every 6th being a passing 16th note, every 12th being a passing eighth note, and every 24th being a passing quarter note.

A master stops the slave simultaneously by sending a MIDI Stop message. The master may then continue to send MIDI Clocks at the rate of its tempo, but the slave should ignore these, and not advance its "song position". Of course, the slave may use these continuing MIDI Clocks to ascertain what the master's tempo is at all times.

Sometimes, a musician will want to start the playback point somewhere other than at the beginning of a song (ie, he may be recording an overdub in a certain part of the song). The master needs to tell the slave what beat to cue playback to. The master does this by sending a Song Position Pointer message. The 2 data bytes in a Song Position Pointer are a 14-bit value that determines the *MIDI Beat* upon which to start playback. Sequences are always assumed to start on a MIDI Beat of 0 (ie, the downbeat). Each MIDI Beat spans 6 *MIDI Clocks*. In other words, each MIDI Beat is a 16th note (since there are 24 MIDI Clocks in a quarter note, therefore 4 MIDI Beats also fit in a quarter). So, a master can sync playback to a resolution of any particular 16th note.

For example, if a Song Position value of 8 is received, then a slave should cue playback to the third quarter note of the song. (8 MIDI beats * 6 MIDI clocks per MIDI beat = 48 MIDI Clocks. Since there are 24 MIDI Clocks in a quarter note, the first quarter occurs on a time of 0 MIDI Clocks, the second quarter note occurs upon the 24th MIDI Clock, and the third quarter note occurs on the 48th MIDI Clock).

A Song Position Pointer message should not be sent while the devices are in play. This message should only be sent while devices are stopped. Otherwise, a slave might take too long to cue its new start point and miss a MIDI Clock that it should be processing.

A MIDI Start always begins playback at MIDI Beat 0 (ie, the very beginning of the song). So, when a slave receives a MIDI Start, it automatically resets its "Song Position" to 0. If the master needs to start playback at some other point (as set by a Song Position Pointer message), then a MIDI Continue message is sent **instead** of MIDI Start. Like a MIDI Start, the MIDI Continue is immediately followed by a MIDI Clock "downbeat" in order to start playback then. The only difference with MIDI Continue is that this downbeat won't necessarily be the very start of the song. The downbeat will be at whichever point the playback was set via a Song Position Pointer message or at the point when a MIDI Stop message was sent (whichever message last occurred). What this implies is that a slave must always remember its "current song position" in terms of MIDI Beats. The slave should keep track of the nearest previous MIDI beat at which it stopped playback (ie, its stopped "Song Position"), in the anticipation that a MIDI Continue might be received next.

Some playback devices have the capability of containing several sequences. These are usually numbered from 0 to however many sequences there are. If 2 such devices are synced, a musician typically may set up the sequences on each to match the other. For example, if the master is a sequencer with a reggae bass line for sequence 0, then a slaved drum box might have a reggae drum beat for sequence 0. The musician can then select the same sequence number on both devices simultaneously by having the master send a Song Select message whenever the musician selects that sequence on the master. When a slave receives a Song Select message, it should cue the new song at MIDI Beat 0 (ie, reset its "song position" to 0). The master should also assume that the newly selected song will start from beat 0. Of course, the master could send a subsequent Song Position Pointer message (after the Song Select) to cue the slave to a different MIDI Beat.

If a slave receives MIDI Start or MIDI Continue messages while it's in play, it should ignore those messages. Likewise, if it receives MIDI Stop messages while stopped, it ignores those.

Ignoring MIDI Messages

A device should be able to "ignore" all MIDI messages that it doesn't use, including currently undefined MIDI messages (ie Status is 0xF4, 0xF5, 0xF9, or 0xFD). In other words, a device is expected to be able to deal with **all** MIDI messages that it could possibly be sent, even if it simply ignores those messages that aren't applicable to the device's functions.

If a MIDI message is not a RealTime Category message, then the way to ignore the message is to throw away its Status and all data bytes (ie, bit #7 clear) up to the next received, non-RealTime Status byte. If a RealTime Category message is received interspersed with this message's data bytes (remember that all RealTime Category messages consist of only 1 byte, the Status), then the device will have to process that 1 Status byte, and then return to the process of skipping the initial message. Of course, if the next received, non-RealTime Status byte is for another message that the device doesn't use, then the "skip procedure" is repeated.

If the MIDI message is a RealTime Category message, then the way to ignore the message is to simply ignore that one Status byte. All RealTime messages have only 1 byte, the Status. Even the two undefined RealTime Category Status bytes of 0xF9 and 0xFD should be skipped in this manner. Remember that RealTime Category messages do **not** cancel running status and also could be sent interspersed with some other message, so any data bytes after a RealTime Category message must belong to some other message.